MICROSOFT®

The High Performance Software™

Microsoft® FORTRAN Optimizing Compiler

User's Guide

MICROSOFT®

The High Performance Software™

Microsoft® FORTRAN Optimizing Compiler

for the MS-DOS® Operating System

User's Guide

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1987

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Microsoft®, the Microsoft logo, MS®, MS-DOS®, and XENIX® are registered trademarks, and CodeView™ is a trademark of Microsoft Corporation.

AT&T® is a registered trademark of American Telephone & Telegraph Company.

DEC® and VAX® are registered trademarks of Digital Equipment Corporation.

IBM® is a registered trademark of International Business Machines Corporation.

Intel® is a registered trademark of Intel Corporation.

Texas Instruments® is a registered trademark of the Texas Instruments Corporation.

Wang® is a registered trademark of Wang Laboratories Incorporated.

Document No. 410500001-400-R06-0187
Part No. 005-014-038

Contents

1 Introduction 1

1.1 Overview 3
1.2 System Requirements 4
1.3 About the Documentation 5
1.4 Notational Conventions 9
1.5 Books about FORTRAN 13
1.6 Reporting Problems 14

2 Getting Started 15

2.1 Introduction 17
2.2 Backing Up Your Disks 17
2.3 Checking the Disk Contents 18
2.4 The SETUP Program 18
2.5 The Compiler Environment 34
2.6 Using an 80186, 80188, or 80286 Processor 40
2.7 Using a RAM Disk 40
2.8 Converting Existing FORTRAN Programs 41
2.9 Quick Start 41
2.10 Practice Session 44
2.11 Using Batch Files 47

3 Compiling: The FL Command 49

3.1 Introduction 51
3.2 The Basics: Compiling, Linking, and Running FORTRAN Files 52
3.3 Using FL Options 58
3.4 Using FL to Link without Compiling 104

4 Linking 107

4.1 Introduction 109
4.2 Running the Linker 109
4.3 Understanding LINK Memory Requirements 110
4.4 Linking with the LINK Command 111

4.5 Linking FORTRAN Program Files 118
4.6 Using Linker Options 121
4.7 Using Overlays 135
4.8 Terminating the LINK Session 137
4.9 How the Linker Works 137

5 Managing Libraries 145

5.1 Introduction 147
5.2 Using LIB: An Overview 148
5.3 Running LIB 149
5.4 Managing Libraries with LIB 157

6 Maintaining Programs with MAKE 163

6.1 Introduction 165
6.2 Using MAKE: An Overview 166
6.3 Creating a MAKE Description File 166
6.4 Maintaining a Program: An Example 170
6.5 Running MAKE 172
6.6 Using MAKE Options 173
6.7 Using Macro Definitions with MAKE 173
6.8 Defining Inference Rules 177

7 Using EXEPACK, EXEMOD, SETENV, and ERROUT 181

7.1 Introduction 183
7.2 The EXEPACK Utility 183
7.3 The EXEMOD Utility 185
7.4 The SETENV Utility 188
7.5 The ERROUT Utility 190

8 Controlling Floating-Point Operations 191

8.1 Introduction 193
8.2 Summary of Math Packages 193
8.3 Selecting Floating-Point Options (/FP) 195

8.4 Using the NO87 Environment Variable 204
8.5 Using Non-IBM®-Compatible Computers 205

9 Working with Memory Models 207

9.1 Introduction 209
9.2 What Is a Memory Model? 211
9.3 FORTRAN Memory Models 215
9.4 Selecting and Adjusting the Memory Model 223

10 Improving Compilation and Execution Efficiency 235

10.1 Introduction 237
10.2 Removing Error-Message Text during SETUP 237
10.3 Compiling and Linking Strategies 237
10.4 Coding Strategies 239

11 Interfaces with Assembly Language and C 245

11.1 Introduction 247
11.2 Assembly-Language Interface 247
11.3 Mixed-Language Programming 266

Appendixes 301

A Differences between Versions 4.0 and 3.3 303

A.1 Introduction 307
A.2 Changes for ANSI Full-Language Standard 307
A.3 Source Compatibility 309
A.4 Object Compatibility 315
A.5 Changes for Version 4.0 317

B Using Exit Codes 335

B.1 Introduction 337
B.2 Exit Codes with MAKE 337
B.3 Exit Codes with DOS Batch Files 338
B.4 Exit Codes for Programs in the FORTRAN Compiler Package 338
B.5 Exit Codes from FORTRAN Programs 341

C Microsoft FORTRAN Record and File Formats 343

C.1 Introduction 345
C.2 Record Structures 345
C.3 Specifying Binary File Format 353

D Handling 8087/80287 Floating-Point Exceptions 355

D.1 Introduction 357
D.2 Controlling the Processing Environment 358
D.3 Reading and Setting Status and Control Values 362

E Error Messages and Limits 365

E.1 Introduction 367
E.2 Command-Line Error Messages 368
E.3 Compiler Error Messages 372
E.4 Run-Time Error Messages 434
E.5 Linker Error Messages 454
E.6 LIB Error Messages 467
E.7 MAKE Error Messages 472
E.8 EXEPACK Error Messages 475
E.9 EXEMOD Error Messages 477
E.10 SETENV Error Messages 479
E.11 ERROUT Error Messages 480
E.12 Compiler and Linker Limits 481

Index 487

Figures

Figure 11.1 Segment Setup in FORTRAN Programs 255
Figure 11.2 C String Stored in Memory 285
Figure 11.3 FORTRAN String Stored in Memory 285
Figure C.1 Formatted Records in Formatted Sequential Files 346
Figure C.2 Formatted Sequential File 346
Figure C.3 Formatted Direct File 347
Figure C.4 Physical Block in Unformatted Sequential File 348
Figure C.5 Logical Record in Unformatted Sequential File 348
Figure C.6 Unformatted Sequential File 349
Figure C.7 Unformatted Direct File 350
Figure C.8 Binary Sequential File 351
Figure C.9 Binary Direct File 352
Figure D.1 Status-Word Format 359
Figure D.2 Control-Word Format 360

Tables

Table 2.1 Organization for 5-1/4-Inch Disks 22

Table 2.2 Organization for 3-1/2-Inch Disks 26

Table 3.1 FL Options and Default Libraries 60

Table 3.2 Default File Names and Extensions 68

Table 3.3 Arguments to Listing Options 69

Table 8.1 Summary of Floating-Point Options 196

Table 9.1 Data Allocation in Large Model 221

Table 9.2 Effects of NEAR Attribute 229

Table 9.3 Effects of FAR and HUGE Attributes 230

Table 11.1 Segments, Groups, and Classes for Standard Memory Models 259

Table 11.2 First Argument Address on Stack for FORTRAN Calling Convention 263

Table 11.3 FORTRAN Return-Value Conventions 264

Table 11.4 Specifying Calling Conventions 268

Table 11.5 Overriding Default Argument-Passing Conventions 269

Table 11.6 Signed 1-Byte Integers 280

Table 11.7 Unsigned 1-Byte Integers 280

Table 11.8 Signed 2-Byte Integers 281

Table 11.9 Unsigned 2-Byte Integers 281

Table 11.10 Signed 4-Byte Integers 282

Table 11.11 Boolean Types 282

Table 11.12 Character Types 283

Table 11.13 Single-Precision Real Numbers 284

Table 11.14 Double-Precision Real Numbers 284

Table 11.15 String and Array Types 285
Table 11.16 Near Pointers 286
Table 11.17 Far Pointers 287
Table 11.18 Procedure Pointers 287
Table 11.19 Arrays 290
Table 11.20 Single-Precision Complex Numbers 291
Table 11.21 Double-Precision Complex Numbers 291
Table 11.22 1-Byte Logical Values 292
Table 11.23 2-Byte Logical Values 292
Table 11.24 4-Byte Logical Values 293
Table A.1 Negative INTEGER or REAL Raised to a REAL Power 312
Table A.2 Zero Raised to a Negative Power 313
Table A.3 COMPLEX Zero Raised to a COMPLEX Power 313
Table A.4 Zero Raised to the Zero Power 314
Table D.1 Floating-Point Exceptions 358
Table D.2 Mask Settings for Operation Exceptions 362
Table E.1 Limits Imposed by the Microsoft FORTRAN Compiler 482
Table E.2 Limits Imposed by the Microsoft Overlay Linker 485

Chapter 1
Introduction

1.1 Overview 3
1.2 System Requirements 4
1.3 About the Documentation 5
1.3.1 About This Manual 6
1.3.2 Finding Information Quickly 8
1.4 Notational Conventions 9
1.5 Books about FORTRAN 13
1.6 Reporting Problems 14

1.1 Overview

The Microsoft® FORTRAN Optimizing Compiler for the MS-DOS® operating system implements the full ANSI standard for the FORTRAN programming language. The FORTRAN language is a powerful general-purpose programming language especially suited to complex scientific, mathematical, engineering, and financial algorithms. The Microsoft FORTRAN Optimizing Compiler provides all of the features required by the ANSI standard, plus many powerful extensions to the standard FORTRAN language.

The Microsoft FORTRAN Optimizing Compiler generates fast, efficient native code. The library that you build using the **SETUP** program provided with the compiler includes code for fast real arithmetic if an 8087/80287 coprocessor is used, or it can provide for software emulation of 8087/80287 operations for systems without a coprocessor. An alternate math package is available to maximize program speed on systems that do not have a coprocessor installed.

The Microsoft FORTRAN Optimizing Compiler also includes the following features:

- Full ANSI 77 FORTRAN
- General Services Administration (GSA) certified error free at Full level
- A full set of intrinsic functions, including standard IBM® VS and DEC® VAX® functions
- The Microsoft CodeView™ debugger, a window-oriented, source-level debugger that makes it easy to find and correct errors in source programs
- A choice of multiple and mixed memory models (medium, large, and huge) to set up the combination of data and code storage that best suits your programs
- Math support, including floating-point emulation, 8087/80287 coprocessor support, and alternate math support for systems without coprocessors
- Large program support
- Extensive diagnostic error messages
- Interlanguage calling support, which allows you to link Microsoft FORTRAN programs with 8086 assembly-language programs and Microsoft C and Pascal programs

- DOS 3.0 networking support, including record and file locking and file sharing
- Compatibility with the XENIX® operating system at the source-code level
- A complete development environment, including the CodeView symbolic debugger, the Microsoft Overlay Linker (**LINK**), the Microsoft Program Maintenance Utility (**MAKE**), and the Microsoft Library Manager (**LIB**)

See Section A.5.4 for a list of the features that have been added to Microsoft FORTRAN for Version 4.0.

Note

Since MS-DOS and PC-DOS are essentially the same operating system, Microsoft manuals use the term DOS to include both systems, except in those cases where a utility (such as **SETENV**) is guaranteed only to work under PC-DOS; in those cases, the term PC-DOS is used explicitly.

1.2 System Requirements

To use the Microsoft FORTRAN Optimizing Compiler, your computer system must have the following components:

- An IBM PC or PC-compatible computer that runs DOS Version 2.0 or later.
- At least two double-sided 5-1/4-inch or 3-1/2-inch disk drives.
- A minimum of 320K (kilobytes) of *available* user memory. (You can determine the available user memory by using the DOS **CHKDSK** utility.)

Note

A hard disk is recommended for this product.

You must use the version of **LINK** included in this package; you cannot use earlier versions of **LINK** with this compiler.

1.3 About the Documentation

This manual explains how to use the Microsoft FORTRAN Optimizing Compiler to compile, link, and run FORTRAN programs on your DOS system. The manual assumes that you are familiar with the FORTRAN language and with DOS, and that you know how to create and edit a FORTRAN-language source file on your system.

If you have questions about the Microsoft FORTRAN language, turn to the *Microsoft FORTRAN Compiler Language Reference*, included in this package. The *Microsoft FORTRAN Compiler Language Reference* includes a glossary, which defines many of the terms used in this documentation. For further reading about FORTRAN, refer to Section 1.5, "Books about FORTRAN." To learn how to use the Microsoft CodeView window-oriented debugger, refer to the Microsoft CodeView manual.

Note

Cross references given in this manual refer to chapters and sections in the *Microsoft FORTRAN Compiler User's Guide*, unless the cross reference is to a chapter or section of another manual in the Microsoft FORTRAN Optimizing Compiler package. In such cases, that manual's title is also given.

1.3.1 About This Manual

The following list describes the remaining chapters of the *Microsoft FORTRAN Compiler User's Guide*:

For Information on:	See:
Organization of the compiler software, how to use the **SETUP** program to install libraries and set up an operating environment for the compiler, and a practice session	Chapter 2, "Getting Started"
Using the basic compiler command **FL** and its most common options for compilation and output	Chapter 3, "Compiling: The FL Command"
The Microsoft Overlay Linker (**LINK**) and the options available for linking FORTRAN program files	Chapter 4, "Linking"
The Microsoft Library Manager (**LIB**), which creates, organizes, and maintains run-time libraries for the Microsoft FORTRAN Optimizing Compiler	Chapter 5, "Managing Libraries"
The Microsoft Program Maintenance Utility (**MAKE**), which updates programs after one or more of their source files are changed	Chapter 6, "Maintaining Programs with MAKE"
The utilities included in the Microsoft FORTRAN Optimizing Compiler package	Chapter 7, "Using EXEPACK, EXEMOD, SETENV, and ERROUT"
The **FL** command options, libraries, and metacommands that determine how your programs handle floating-point math	Chapter 8, "Controlling Floating-Point Operations"

Definitions of memory models, standard memory models offered with the Microsoft FORTRAN Optimizing Compiler, and how to choose and adjust the standard memory models to improve program efficiency and accommodate large data items	Chapter 9, "Working with Memory Models"
Installing, coding, and compiling strategies that help reduce the size of program executable files and the amount of memory they require	Chapter 10, "Improving Compilation and Execution Efficiency"
The interface between assembly-language routines and FORTRAN routines, and mixed-language programming using the Microsoft FORTRAN and Microsoft C Compilers	Chapter 11, "Interfaces with Assembly Language and C"
Differences between Versions 4.0 and 3.3 of the Microsoft FORTRAN Compiler and instructions for converting programs written for versions prior to 4.0 to the format accepted by Version 4.0	Appendix A, "Differences between Versions 4.0 and 3.3"
Exit codes produced by each of the programs in the Microsoft FORTRAN Optimizing Compiler package and use of exit codes in **MAKE** description files and DOS batch files	Appendix B, "Using Exit Codes"
The record structure in files created by the Microsoft FORTRAN Optimizing Compiler	Appendix C, "Microsoft FORTRAN Record and File Formats"
How Microsoft FORTRAN deals with floating-point exceptions generated by the 8087 and 80287 coprocessors	Appendix D, "Handling 8087/80287 Floating-Point Exceptions"

The error messages generated by the FORTRAN compiler, linker, utilities, and other programs in the Microsoft FORTRAN Optimizing Compiler package; run-time error messages produced by executable programs written in FORTRAN; and compiler and linker limits	Appendix E, "Error Messages and Limits"

1.3.2 Finding Information Quickly

The following list gives guidelines for finding information quickly in the Microsoft FORTRAN documentation:

- If you know which option or group of options you want to use and you just need a refresher about its form or purpose, the *Microsoft FORTRAN Compiler Quick Reference Guide* is your best source.
- If you need more complete information than is found in the *Microsoft FORTRAN Compiler Quick Reference Guide*, use the index at the back of this manual. You can look up options under their names (for example, **/Fl**) or under their functions (for example, "Listing files" or "Object-listing files"). Each manual in this package has its own index.
- When you look up an option in this manual's index, you may want to make a note of the page number next to the option in the *Microsoft FORTRAN Compiler Quick Reference Guide*. Next time you look up the option in the *Microsoft FORTRAN Compiler Quick Reference Guide*, you won't need to use the index but can turn directly to the given page in this manual if you need more information.
- You can get a general sense of what you can do with the compiler and utilities and what topics are covered in this manual by browsing through the table of contents at the beginning of this manual or the chapter outlines at the beginning of each chapter.

1.4 Notational Conventions

This manual uses the notational conventions described in the following list:

Example of Convention	Description of Convention
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen or by your printer. For example, the following command line is printed in this special typeface: `FL /Fs /St'Main Title' FILE.FOR` When discussing this command line in text, items appearing on the command line, such as `'Main Title'`, also appear in the special typeface.
`User input`	This darker typeface shown in the left column indicates user input in examples that include both user input and program output, as shown in the following example: `Object Modules [.OBJ]: FUN TEXT` This example shows the **LINK** prompt `Object Modules [.OBJ]:`, and the user input **`FUN TEXT`**.
placeholders	Words in italics are placeholders for types of information that you must supply. A file name is an example of this kind of information. In the following statement, *objfile* is italicized to show that this is a general form for the **LINK** command: **LINK** *objfile*; In an actual program statement, the placeholder *objfile* must be replaced by a specific object-file name, as in the following example: `LINK TEST.OBJ;` Italics are also occasionally used in the text for emphasis.

KEYWORDS and other concepts

Bold capital letters indicate commands, FORTRAN keywords, and the names of internal and external files associated with the Microsoft FORTRAN Optimizing Compiler. They also indicate DOS commands, reserved words, and internal names.

In the following example, **FL** is the name of a command, and **/Zi** is the name of a compiler option:

FL /Zi *filename*

other keywords

Bold lowercase letters are sometimes used to indicate keywords of other languages.

In the sentence, "The value that is returned by **LOCNEAR** is equivalent to a **near** function or data pointer in Microsoft C," the word **LOCNEAR** is a FORTRAN keyword, and the word **near** is a keyword of Microsoft C.

[(, * / =)]

Bold type indicates any punctuation or symbols (such as commas, parentheses, semicolons, hyphens, equal signs, and operators) that you must type exactly as shown.

For example, the following illustrates the syntax for macros in description files for the **MAKE** utility:

$(*name*)

This means you must type the characters **$(**, then the macro name, then the character).

Apostrophes: **''**

An apostrophe is entered as a single right quotation mark (’), not a single left quotation mark (‘). Note that in the typeface used in examples, such as `'string'`, apostrophes look like this: `'`.

[[optional items]]

Double square brackets surround anything that is optional.

The following example shows that in the **LINK** option **/DSALLOCATE**, the string **ALLOCATE** is optional:

/DS[[ALLOCATE]]

Thus, either of the following **LINK** commands is acceptable:

```
LINK /DS TEST.OBJ;

LINK /DSALLOCATE TEST.OBJ;
```

Note

Double square brackets ([[ ]]) are a syntax convention used in this manual to indicate optional items. Single square brackets ([ ]) are punctuation that should be typed where shown.

{choice1 | choice2}

Braces and a vertical bar indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items unless all of the items are also enclosed in double square brackets.

For example, the **/W** (warning-level) compiler option has the following syntax:

/W {0 | 1}

You can use **/W1** to display warning messages or **/W0** to suppress them.

"Defined term"

Quotation marks set off terms defined in the text. For example, the term "far" appears in quotation marks the first time it is defined.

Quotation marks also set off command-line prompts in text. For example, **LINK** prompts you for names of object files; this prompt is called the "Object Modules" prompt.

Some FORTRAN command-line options require quotation marks. Quotation marks that are required by the language are shown in the form

" "

rather than " " (as they would appear in text). For example, the **/V** option has the following form:

/V"*string***"**

Repeating elements...

Three dots following an item indicate that more items having the same form may be entered.

For example, the syntax of the **/I** compiler option is as follows:

/I *directory* [[**/I** *directory...*]]

The dots following **/I** *directory* indicate that you can enter more than one **/I** option followed by directory names on the **FL** command line.

```
Program
.
.
.
Fragment
```

A column of dots in syntax lines and program examples shows that a portion of the program has been omitted.

For instance, in the following program fragment, only two lines are shown, and the lines in between are omitted:

```
CALL getnum(I,*10)
.
.
.
SUBROUTINE getnum(I,*)
```

KEY NAMES	Small capital letters are used for the names of keys and key sequences, which you must press. Examples include CONTROL-C and ENTER.

■ Example

The following example shows how this manual's notational conventions are used to indicate the syntax of the **FL** command:

FL [[*option...*]] [[*filespec...*]] [[*option...*]] [[*filespec...*]] [[**/link**[[*libfield*]] [[*linkoptions*]]]

This syntax listing shows that, when using the **FL** command, you must first enter the **FL** command. Then you can optionally enter any number of options (*option*) and source- or object-file specifications (*filespec*) before the optional **/link** field. If you use the **/link** field, it can optionally be followed by a library name (*libfield*) and one or more linker options (*linkoptions*).

1.5 Books about FORTRAN

The following books contain information on FORTRAN programming:

Agelhoff, Roy, and Richard Mojena. *Applied FORTRAN 77, Featuring Structured Programming*. Belmont, Calif.: Wadsworth, 1981.

Ashcroft, J., R. H. Eldridge, R. W. Paulson, and G. A. Wilson. *Programming with FORTRAN 77*. Dobbs Ferry, N.Y.: Sheridan House, Inc., 1981.

Friedman, Frank, and E. Koffman. *Problem Solving and Structured Programming in FORTRAN*. 2d ed. Reading, Mass.: Addison-Wesley, 1981.

Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*. New York, N.Y.: McGraw-Hill, 1978.

Wagener, Jerrold L. *FORTRAN 77: Principles of Programming*. New York, N.Y.: John Wiley and Sons, Inc., 1980.

These books are listed for your convenience only. Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

1.6 Reporting Problems

If you need help or you feel you have discovered a problem in the software, use the Software Problem Report form at the back of this manual to send this information to Microsoft Corporation. Please provide the following information:

- The compiler version number (from the logo that is printed when you invoke the compiler with **FL**)
- The version of DOS you are running (use the DOS **VER** command)
- Your system configuration (type of machine you are using and its total memory, total free memory at compiler execution time, and any other information you think might be useful)
- The command line used in the compilation
- Any nonstandard object files or libraries needed to link, in addition to the standard object files or libraries you linked with at the time the problem occurred

If your program is very large, please try to reduce its size to the smallest possible program still producing the problem.

If you have comments or suggestions regarding any of the manuals accompanying this product, please use the Documentation Feedback reply card at the back of this manual to send them to Microsoft Corporation.

Getting Started

Chapter 2
Getting Started

2.1 Introduction 17
2.2 Backing Up Your Disks 17
2.3 Checking the Disk Contents 18
2.4 The SETUP Program 18
2.4.1 What SETUP Does 18
2.4.2 Starting SETUP 19
2.4.3 Installing on a Hard-Disk System 19
2.4.4 Installing on a 5-1/4-Inch Floppy-Disk System 21
2.4.5 Installing on a 3-1/2-Inch Floppy-Disk System 25
2.4.6 SETUP Library Options 29
2.4.6.1 Choosing between Medium- and Large-Model Libraries 29
2.4.6.2 Choosing a Math Package 29
2.4.6.3 Naming Libraries 31
2.4.6.4 Removing Error-Message Text 32
2.4.6.5 Compatibility with Microsoft C 32
2.4.6.6 Compatibility with Versions 3.2 and 3.3 of Microsoft® FORTRAN 33
2.4.6.7 Running SETUP More than Once 33
2.5 The Compiler Environment 34
2.5.1 Environment Variables 35
2.5.1.1 The PATH Variable 36
2.5.1.2 The LIB Variable 36
2.5.1.3 The INCLUDE Variable 36

2.5.1.4 The TMP Variable 37
2.5.1.5 Setting Environment Variables 37
2.5.2 CONFIG.SYS Settings 39
2.6 Using an 80186, 80188, or 80286 Processor 40
2.7 Using a RAM Disk 40
2.8 Converting Existing FORTRAN Programs 41
2.9 Quick Start 41
2.9.1 Quick Overview 42
2.9.2 Simple Compile and Link 42
2.9.3 Using Wild Cards 42
2.9.4 Compiling without Linking 43
2.9.5 Using the Emulator Library 43
2.9.6 Preparing to Use the CodeView Debugger 43
2.9.7 Using the Debug and Declare Options 44
2.9.8 Setting Titles and Subtitles 44
2.9.9 Compiling a Free-Form File 44
2.10 Practice Session 44
2.11 Using Batch Files 47

2.1 Introduction

This chapter explains how to use the **SETUP** program to install the Microsoft FORTRAN Compiler software and set up an operating environment for the compiler.

To get your FORTRAN compiler up and running as quickly as possible, use the following procedures:

1. Back up your disks (see Section 2.2).
2. Check the contents of the disks (see Section 2.3).
3. Read the **README.DOC** file to learn about changes and additions made to the software after this manual was printed.
4. Run the **SETUP** program to install the software.
5. Read Section 2.9, "Quick Start," or Section 2.10, "Practice Session," to learn how to compile and link.

Important

Step 4 is required. You must run the **SETUP** program before you can use the libraries provided with the Microsoft FORTRAN Compiler.

Several DOS procedures are mentioned in this chapter. In particular, the DOS **SET** and **PATH** commands are used to give values to "environment variables," which control the compiler environment. If you are unfamiliar with the **SET** and **PATH** commands, or with other DOS procedures mentioned in this chapter, consult your DOS documentation for instructions.

2.2 Backing Up Your Disks

The first thing you should do after removing your system disks from the disk packet included with the Microsoft FORTRAN Compiler is to make working copies, using the DOS **COPY** command or the **DISKCOPY** utility. Save the original disks for making future working copies.

2.3 Checking the Disk Contents

The Setup distribution disk in your compiler package contains a file named **PACKING.LST**. This file lists and describes the files that make up the compiler software. It also lists the manuals and other materials included in the package that help you use the software.

Use the **PACKING.LST** file to get a quick overview of the compiler software and verify that you have a complete package.

Note

Named disks included with the Microsoft FORTRAN Compiler package are referred to as distribution disks to distinguish them from disks you create and label as you use the **SETUP** program.

2.4 The SETUP Program

The **SETUP** program automatically installs the compiler software. You will find the **SETUP** program on the Setup distribution disk (the disk may contain other files as well). The following sections explain what **SETUP** does and how to start **SETUP**.

2.4.1 What SETUP Does

The **SETUP** program performs the following tasks:

- Copies all necessary files to the directories or disks you specify and creates back-up copies of existing files that would otherwise be overwritten.
- Builds a run-time library based on your specifications. This library includes support for the math, memory-model, and compatibility options you choose when you run **SETUP**. Under many circumstances, this is the only library you will need when you link.

See the **PACKING.LST** file on the Setup distribution disk for a complete list of the files provided with the Microsoft FORTRAN Compiler. (This list includes the files that the **SETUP** program installs and uses to build libraries.) See Section 2.5, "The Compiler Environment," for more information on environment variables and the **CONFIG.SYS** file.

2.4.2 Starting SETUP

To start **SETUP**, do the following:

1. Insert the Setup distribution disk in a floppy-disk drive.
2. Make the drive in which you inserted the Setup distribution disk the current drive.
3. Type

   ```
   SETUP
   ```

   If you would like to practice using **SETUP** without actually installing the compiler software, type

   ```
   SETUP /n
   ```

   The **/n** (for "No-op") option allows you to become familiar with the program's operation before you use it to perform an actual installation.

The **SETUP** program guides you through the installation process, allowing you to install the software on either a hard-disk or a floppy-disk system. After you install the compiler software, you must change the values of your environment variables so that the compiler and linker can find the files they need. **SETUP** displays a screen showing the values that you must assign to each environment variable.

You can run the **SETUP** program without reading any further in this section, since **SETUP** provides all the information you need. However, you may find the information in the following sections helpful.

2.4.3 Installing on a Hard-Disk System

If you install the compiler software on a hard disk, **SETUP** installs the compiler passes, libraries, and utilities in subdirectories on the hard disk. If these subdirectories do not exist, **SETUP** creates them. It also sets up a

subdirectory to be used for the temporary files that are created during compilation. The following list shows the default subdirectory that **SETUP** uses for each type of file; however, you can tell **SETUP** to use a subdirectory other than the default:

Default Subdirectory	Files
\BIN	Compiler passes, linker, utilities, and error-message files
\TMP	Temporary files
\LIB	Library files

SETUP also allows you to install a mouse driver—either **MOUSE.SYS** or **MOUSE.COM**. The mouse driver allows you to use a mouse with the Microsoft CodeView window-oriented debugger. If you choose to install a mouse driver, **SETUP** installs the mouse driver in the **\MOUSE1** subdirectory by default. You must make sure that your environment is set up correctly for the mouse driver and make any necessary changes to environment variables so that the mouse driver can be found.

Note

SETUP does not install the following files:

- Source programs, including the **DEMO.FOR** demonstration program on the Utilities and Source Code distribution disk.

  If you want to install the **DEMO.FOR** program or other source programs, create a subdirectory to hold them or use an existing directory. Then copy source programs to this subdirectory before compiling them.

- Files on the Learning Microsoft CodeView distribution disk.

  You can run the tutorial for the Microsoft CodeView debugger directly from the distribution disk. (See the Microsoft CodeView manual for more information.)

- Certain special-purpose files. These files are discussed in the **README.DOC** file on the Setup distribution disk.

After you run **SETUP**, you must set or change the values of your environment variables to reflect the directories used for the compiler, library, and temporary files so that the compiler can find the files it needs. **SETUP** displays a screen indicating the values you should assign to these variables. One simple way to assign these values is to use a batch file, as described in Section 2.11. You can also change the values assigned to environment variables in your **AUTOEXEC.BAT** file. If you make the changes in **AUTOEXEC.BAT**, the compiler environment is set up automatically every time you reboot. (See Section 2.5 for a description of environment variables and the FORTRAN compiler environment.)

Besides changing the values of environment variables, you may have to change the settings in your **CONFIG.SYS** file (see Section 2.5.2 for more information).

After you have copied your source programs to a subdirectory, installed the compiler software, and changed your environment as needed, use the following procedure to compile programs:

1. Use the DOS **CD** command to make the directory with your source programs the current working directory.
2. Type an **FL** command line to start compiling. (See Chapter 3 for a description of the options that you can specify on the **FL** command line to control the compilation process.)

2.4.4 Installing on a 5-1/4-Inch Floppy-Disk System

You can install the Microsoft FORTRAN Compiler on 5-1/4-inch floppy disks if your system has two 5-1/4-inch floppy-disk drives.

Before you install the compiler software on a 5-1/4-inch floppy-disk system, format at least six blank disks. **SETUP** uses five blank disks for the compiler software and libraries. In addition, you should copy source programs, including the **DEMO.FOR** demonstration program on the Utilities and Source Code distribution disk, to the sixth blank disk.

If you request compatibility with Versions 3.2 and 3.3 of Microsoft FORTRAN, **SETUP** requires a seventh blank disk (the Compatibility disk); see the discussion following the list of disk contents in Table 2.1.

In addition to the blank disks, **SETUP** may ask you to provide the DOS disk you normally use to boot your system. **SETUP** installs the mouse driver, if you request one, on this disk.

When **SETUP** installs the compiler software on your disks, it organizes files so that you can conveniently carry out tasks in sequence.

SETUP prompts you to exchange the disks in the destination drive when it has finished copying the appropriate files to that disk. When **SETUP** finishes, your disks will be organized as shown in Table 2.1.

Table 2.1

Organization for 5-1/4-Inch Disks

Disk	Files	Contents of Files
Driver	**F1.EXE**	Pass 1 of the compiler
	F1.ERR	Error-message file for Pass 1 of the compiler
	FL.EXE	Executable file for the **FL** command
	FL.HLP	Help file for the **FL** command
	FL.ERR	Error-message file for the **FL** command
Compiler	**F2.EXE**	Pass 2 of the compiler
	F3.EXE	Pass 3 of the compiler
	F23.ERR	Error-message file for Passes 2 and 3 of the compiler
Link	**F23.ERR**	Error-message file for alternate Pass 3
	F3S.EXE	Alternate Pass 3 of the compiler. This pass is used if optimization is disabled during compiling. (See Section 3.3.15, "Optimizing," for more information.)
	LINK.EXE	Executable file for the linker
	*x*LIBFOR*x*.LIB	Library created by **SETUP**

Table 2.1 *(continued)*

Disk	Files	Contents of Files
Utility	**LIB.EXE**	Executable file for the **LIB** utility
	MAKE.EXE	Executable file for the **MAKE** utility
	EXEPACK.EXE	Executable file for the **EXEPACK** utility
	EXEMOD.EXE	Executable file for the **EXEMOD** utility
	ERROUT.EXE	Executable file for the **ERROUT** utility
	SETENV.EXE	Executable file for the **SETENV** utility
	CV.EXE	Executable file for the CodeView debugger
	CV.HLP	Help file for the CodeView debugger
Scratch	**LIB.EXE**	Executable file for the **LIB** utility, which is used to help build the run-time library. **SETUP** installs this file on this disk.
	Intermediate versions of the libraries	Present while **SETUP** is running; deleted from the disk if **SETUP** finishes normally

If you want compatibility with Versions 3.2 and 3.3 of Microsoft FORTRAN, **SETUP** copies the "compatibility" library, **FORTRAN.LIB**, to one of your disks. You must provide **SETUP** with a disk that has enough room to hold **FORTRAN.LIB**. This disk can be either a formatted blank disk or a disk with earlier-version object files that you want to link. Label this disk the Compatibility disk before you provide it to **SETUP**.

Note

SETUP does not install the following files:

- Files on the Learning Microsoft CodeView distribution disk.

  You can run the tutorial for the Microsoft CodeView debugger directly from the distribution disk. (See the Microsoft CodeView manual for more information.)

- Certain special-purpose files. These files are discussed in the **README.DOC** file on the Setup distribution disk.

After you run **SETUP**, you must set or change the values of your environment variables to reflect the disks used for the compiler, library, and temporary files, so that the compiler can find the files it needs. **SETUP** displays a screen indicating the values you should assign to these variables. One simple way to assign these values is to use a batch file, as described in Section 2.11. You can also change the values assigned to environment variables in your **AUTOEXEC.BAT** file. If you make the changes in **AUTOEXEC.BAT**, the compiler environment is set up automatically every time you reboot. (See Section 2.5 for a discussion of environment variables and the FORTRAN compiler environment.)

Besides changing the values of environment variables, you may have to change the settings in your **CONFIG.SYS** file; see Section 2.5.2 for more information.

After you have copied your source programs to a disk, installed the compiler software, and changed your environment as needed, use the following procedure to compile programs:

1. Insert the Driver disk in Drive A.
2. Insert the disk containing your source program in Drive B.
3. Type the following DOS command to make Drive B the current drive:

   ```
   B:
   ```

4. Type an **FL** command line to start compiling. (See Chapter 3 for a description of **FL** command-line options that control compilation.)
5. When prompted, swap the Compiler, Link, and (if required) Compatibility disks in Drive A.

Notes

If you prefer, you can reverse the uses of the two drives. In this case, you would insert the source disk into Drive A and swap the Driver, Compiler, and Link disks in and out of Drive B. If you want to reverse the uses of the drives, be sure that you change the settings of your environment variables accordingly.

After the compiling and linking process has finished, the executable file will be on the disk most recently used in Drive B (or Drive A if you choose to reverse drives). Ordinarily, this is the disk containing your source program. However, if you install **FORTRAN.LIB** on a disk containing object files compiled with Version 3.2 or 3.3 and then link with that library, the executable file may end up on the disk containing **FORTRAN.LIB** and the object files.

2.4.5 Installing on a 3-1/2-Inch Floppy-Disk System

You can install the Microsoft FORTRAN Compiler on 3-1/2-inch floppy disks if your system has two 3-1/2-inch floppy-disk drives.

Before you install the compiler software on a 3-1/2-inch floppy-disk system, format at least five blank disks. Copy source programs, including the **DEMO.FOR** demonstration program on the Utilities, Source Code, and Microsoft CodeView distribution disk, to one of the blank disks. **SETUP** uses the remaining four blank disks for the compiler software and libraries.

In addition to the blank disks, **SETUP** may ask you to provide the DOS disk you normally use to boot your system. **SETUP** installs the mouse driver, if you request one, on this disk.

When **SETUP** installs the compiler software on your disks, it organizes files so that you can conveniently carry out tasks in sequence. **SETUP** prompts you to exchange the disks in the destination drive when it has finished copying the appropriate files to that disk.

When **SETUP** finishes, your disks will be organized as shown in Table 2.2.

Table 2.2

Organization for 3-1/2-Inch Disks

Disk	Files	Contents of Files
Driver/Compiler	**F1.EXE**	Pass 1 of the compiler
	F1.ERR	Error-message file for Pass 1 of the compiler
	FL.EXE	Executable file for the **FL** command
	FL.HLP	Help file for the **FL** command
	FL.ERR	Error-message file for the **FL** command
	F2.EXE	Pass 2 of the compiler
	F3.EXE	Pass 3 of the compiler
	F23.ERR	Error-message file for Passes 2 and 3 of the compiler and for alternate Pass 3 (see above)
	F3S.EXE	Alternate Pass 3 of the compiler. This pass is used if optimization is disabled during compiling. (See Section 3.3.15, "Optimizing," for more information.)
Link	**LINK.EXE**	Executable file for the linker
	***x*LIBFOR*x*.LIB**	Library created by **SETUP**
	FORTRAN.LIB	"Compatibility" library; present if you choose compatibility with Versions 3.2 and 3.3 of the Microsoft FORTRAN Compiler

Table 2.2 *(continued)*

Disk	Files	Contents of Files
Utility	**LIB.EXE**	Executable file for the **LIB** utility
	MAKE.EXE	Executable file for the **MAKE** utility
	EXEPACK.EXE	Executable file for the **EXEPACK** utility
	EXEMOD.EXE	Executable file for the **EXEMOD** utility
	ERROUT.EXE	Executable file for the **ERROUT** utility
	SETENV.EXE	Executable file for the **SETENV** utility
	CV.EXE	Executable file for the CodeView debugger
	CV.HLP	Help file for the CodeView debugger
Scratch	**LIB.EXE**	Executable file for the **LIB** utility, which is used to help build the run-time library. **SETUP** installs this file on this disk.
	Intermediate versions of the libraries	Present while **SETUP** is running; deleted from the disk if **SETUP** finishes normally

Note

SETUP does not install the following files:

- Files on the Learning Microsoft CodeView distribution disk.

  You can run the tutorial for the Microsoft CodeView debugger directly from the disk provided in the package. (See the Microsoft CodeView manual for more information.)

- Certain special-purpose files. These files are discussed in the **README.DOC** file on the Setup and Compiler distribution disk.

After you run **SETUP**, you must set or change the values of your environment variables to reflect the disks used for the compiler, library, and temporary files, so that the compiler can find the files it needs. **SETUP** displays a screen indicating the values you should assign to these variables. One simple way to assign these values is to use a batch file, as described in Section 2.11. You can also change the values assigned to environment variables in your **AUTOEXEC.BAT** file. If you make the changes in **AUTOEXEC.BAT**, the compiler environment is set up automatically every time you reboot. (See Section 2.5 for a discussion of environment variables and the FORTRAN compiler environment.)

Besides changing the values of environment variables, you may have to change the settings in your **CONFIG.SYS** file; see Section 2.5.2 for more information.

After you have copied source programs to a disk, installed the compiler software, and set up or changed your environment as needed, use the following procedure to compile programs:

1. Insert the Driver/Compiler disk in Drive A.
2. Insert the disk containing the source program in Drive B.
3. Type the following DOS command to make Drive B the current drive:

   ```
   B:
   ```

4. Type an **FL** command line to start compiling. (See Chapter 3 for a description of the **FL** command-line options that control compilation.)
5. When prompted, insert the Link disk in Drive A.

Note

If you prefer, you can reverse the uses of the two disk drives. In this case, you would insert the source disk into Drive A and swap the Driver/Compiler and Link disks in and out of Drive B. If you want to reverse the use of the drives, be sure that you change the settings of your environment variables accordingly.

2.4.6 SETUP Library Options

SETUP offers you several options to customize the FORTRAN library it creates for your programs. See Sections 2.4.6.1 – 2.4.6.7 for a description of these options.

Note

The choices you make when you first run **SETUP** do not have to be permanent. You can always run **SETUP** again to build and install additional libraries.

2.4.6.1 Choosing between Medium- and Large-Model Libraries

The **SETUP** program allows you to create and install either a medium- or a large-model library. You can run **SETUP** more than once to create and install both versions if you have enough room.

If you are unfamiliar with memory models or do not care which memory model you will be using, or if you will be linking with object files compiled with Version 3.2 or 3.3 of the Microsoft FORTRAN Compiler, choose a large-model library. If you do not tell the compiler which memory model to use, it uses the large memory model, which requires a large-model library. If you are linking with Version 3.2 or Version 3.3 object files, you must use a large-model library. Also, the huge memory model uses large-model libraries.

If you want to use the medium model, you must install a medium-model library, then compile your source program with the appropriate **FL** command-line option. See Chapter 9 for a discussion of memory models.

2.4.6.2 Choosing a Math Package

You can choose among three different packages to handle floating-point math operations:

1. The 8087/80287 package (the default)
2. The emulator package
3. The alternate math package

Your choice depends both on whether you have an 8087 or 80287 coprocessor and on the kind of development work you are doing. If you have an 8087 or 80287 coprocessor installed, you can use the default math package. If you don't have an 8087 or 80287 coprocessor, you must choose the emulator or alternate math package, then use the appropriate **FL** command-line option when you compile the program to select the appropriate library for that package.

Note

If you're not sure which math package you want to use and you have a coprocessor installed, choose the default (8087/80287) package. If you're not sure and you do not have a coprocessor installed, choose the emulator package. If you choose the emulator package, remember that you must give an **/FP** option on the command line when you compile programs; see Section 3.3.1, "Floating-Point and Memory-Model Options," and Chapter 8, "Controlling Floating-Point Operations," for more information.

The following list briefly summarizes the three math packages:

- The 8087/80287 package requires an 8087 or 80287 coprocessor. This is the default package.
- The emulator package provides most of the functions of a coprocessor in software. Install this package if you don't have a coprocessor and you want very accurate floating-point results. If you have a coprocessor and choose the emulator package, the coprocessor is used.
- The alternate math package gives you more speed than the emulator but less accuracy; install it if you don't have a coprocessor and speed is more important than accuracy. If you have a coprocessor and choose the alternate math package, the coprocessor is not used.

If you have enough space on your hard disk, you may want to build and install additional libraries that support different math packages. This gives you the flexibility to change libraries easily.

See Chapter 8 for more information about floating-point operations.

2.4.6.3 Naming Libraries

SETUP gives the library it builds a default name based on the memory model and math package you choose. The default name has the following form:

{L | M}LIBFOR{7 | E | A}.LIB

The first character of the default library name is determined by the memory model you choose: "L" if you choose the large (default) or huge memory model and "M" if you choose the medium memory model.

The last character of the default library base name is determined by the math package you choose: "7" if you choose the 8087/80287 math package (the default), "E" if you choose the emulator package, or "A" if you choose the alternate math package.

Note that if you choose the default options for both the memory model and the math package, **SETUP** gives the library it builds the default name **LLIBFOR7.LIB**.

Sections 2.4.6.4 – 2.4.6.6 describe three additional options that you can choose when you build a library. The names of the libraries generated when you choose these options are the same as the names given above; however, you can rename libraries to distinguish them, provided you explicitly specify the new library name at link time. (If you do not give the linker the new name, the linker expects that the library name is the default for the floating-point and memory-model compiler options you have chosen.)

Note

For ease of discussion, the remainder of this manual uses the default names to identify libraries that support particular combinations of memory models and math packages.

2.4.6.4 Removing Error-Message Text

SETUP allows you to choose how run-time error messages generated by your program are handled: you can choose to display both an error number and an error message, or you can choose just to display an error number. In either case, you can look up the message text and explanation in Appendix E, "Error Messages and Limits." The advantage to removing error-message text is that it reduces by approximately 2K the size of the executable files that the compiler creates.

This option does not affect the text of math errors or start-up and termination errors. For these errors, message text is still displayed.

2.4.6.5 Compatibility with Microsoft C

A powerful feature of the Microsoft FORTRAN Compiler is its compatibility with Microsoft C: you can mix Microsoft FORTRAN and Microsoft C modules in the same program. This means that you can use Microsoft C modules to perform operations, such as system-level operations, that would be difficult to write in FORTRAN.

If you plan to do mixed-language programming, you include compatibility with Microsoft C in the library that **SETUP** builds. You must use this library when you link FORTRAN and C object files. The FORTRAN library that is compatible with Microsoft C leaves out certain low-level C routines that are required for linking C modules and assumes that these routines are taken from the C library.

Note that the standard C library must also be present when you use this library, even if you are linking only FORTRAN modules. See Section 11.3.12.3, "Linking Considerations," for information about library requirements when linking FORTRAN and C modules.

Note

Use this library only for mixed-language programs. It is not suitable for programs made up entirely of Microsoft FORTRAN modules.

2.4.6.6 Compatibility with Versions 3.2 and 3.3 of Microsoft® FORTRAN

If you are upgrading from Version 3.2 or Version 3.3 of the Microsoft FORTRAN Compiler and plan to link with object files compiled under these earlier versions, indicate at the **SETUP** prompt that you want compatibility with Versions 3.2 and 3.3. In this case, **SETUP** creates a library that you can use when linking object files or libraries created under Versions 3.2 and 3.3.

The only difference between this library and the corresponding library that does not include compatibility is that the Version 3.2/3.3 compatible library is larger. You can use the Version 3.2/3.3 compatible library for Version 4.0 object files.

A library that is compatible with Versions 3.2 and 3.3 contains the library interfaces that ensure that math operations are handled correctly between the Version 4.0 object files and the earlier object files. Thus, you do not need to use math libraries from the earlier versions for linking.

In addition to the Version 3.2/3.3 compatible library, **SETUP** installs a "compatibility" library named **FORTRAN.LIB** on the disk. You must use **FORTRAN.LIB** in addition to the library when you link with object files created under Versions 3.2 and 3.3.

If you have told **SETUP** to create a medium-model library, it does not ask if you want compatibility with Versions 3.2 and 3.3, since only the large memory model is supported in versions of Microsoft FORTRAN prior to Version 4.0.

See Appendix A for more information about compatibility between Versions 4.0 and 3.3.

2.4.6.7 Running SETUP More than Once

You can rerun the **SETUP** program to build custom versions of the FORTRAN library after the initial installation. For example, if you want to have libraries that support each of the math packages, you can run **SETUP** after you initially install the compiler and library software to create the additional libraries.

If you have installed the compiler software on a hard disk, use the following procedure to run **SETUP** again from your hard disk:

1. Find or create a subdirectory to hold the library component files.
2. Copy **SETUP.EXE** and **SETUP.DAT** from the Setup distribution disk, and all files from the Large-Model and Medium-Model Libraries distribution disks, to the subdirectory you created or found in Step 1. Note that identical copies of certain files appear on both disks, so there is no harm in overwriting these duplicate files.
3. Execute the DOS **CD** command to make the directory to which you copied **SETUP.DAT**, **SETUP.EXE**, and the library files your current working directory.
4. Type

   ```
   SETUP
   ```

   to rerun **SETUP**.

Alternately, you can place **SETUP.EXE** in the **\BIN** subdirectory, if **\BIN** is specified by the **PATH** environment variable, or in another subdirectory specified by the **PATH** environment variable. However, you must leave **SETUP.DAT** in the subdirectory to which you copied the library components.

If you have installed the compiler software on floppy disks, simply format new blank disks to hold the new libraries and rerun the **SETUP** program as usual to create and install the new libraries.

2.5 The Compiler Environment

The compiler environment consists of a set of environment variables that tells the compiler and linker where to find the files they need to process a program. They are called environment variables because they define the environment in which the compiler and linker operate. Environment variables are defined at the DOS command level using the DOS commands **SET** and **PATH**.

SETUP proposes default environment settings based on where you tell it to install various types of files. After you run **SETUP**, you can include these environment settings in your **AUTOEXEC.BAT** file to ensure that the compiler environment is set up properly every time you reboot.

In addition to changing **AUTOEXEC.BAT**, you may have to change your **CONFIG.SYS** file, as described in Section 2.5.2, so that it satisfies the compiler requirements.

The following sections describe the environment variables and the **CONFIG.SYS** settings used by the compiler and linker.

2.5.1 Environment Variables

The compiler and linker use four environment variables: **PATH**, **LIB**, **TMP**, and **INCLUDE**. Each of these variables is assigned one or more path specifications (in the case of **TMP**, only one path specification) that tell the compiler or linker where to find a particular type of file, as shown in the following list:

Environment Variable	Type of File
PATH	Executable files (any file ending with **.EXE**). These files include the compiler control program (**FL.EXE**), the compiler passes, the linker, all utilities and **.HLP** and **.ERR** files.
LIB	Library files (any file ending with **.LIB**).
TMP	Temporary files created by the compiler; only one path specification may be used.
INCLUDE	Include files.

Note

The compiler or linker always searches the current working directory first before searching the locations given in an environment variable. For include files, the compiler searches the directory of the file containing the **$INCLUDE** metacommand.

Although environment variables are usually helpful, you are not required to set them. If you do not set these variables, the current working directory is used to search for files and create temporary files.

2.5.1.1 The PATH Variable

DOS uses the **PATH** setting to locate executable files. The **PATH** setting contains one or more path specifications, separated by semicolons, that give the locations of executable files.

By setting **PATH** to the path specification of the directory containing **FL.EXE**, you can execute the **FL** command from any directory. **FL** uses the **PATH** setting to locate the compiler passes and the linker. Thus, the **PATH** setting should also include the directories where those files are located.

2.5.1.2 The LIB Variable

The **LIB** environment variable defines where the linker searches for libraries. (Section 4.5.3 gives the rules the linker follows when searching for libraries.) This variable can contain one or more path specifications, separated by semicolons.

When you compile a source file using the Microsoft FORTRAN Compiler, the compiler places a library name in the object file it creates. This name is the name of the library that supports the memory-model and floating-point options you have given on the **FL** command line.

The linker searches the standard places for this library. The linker also uses the **LIB** setting to search for any other libraries that you specify on the command line at link time. See Section 4.5.3 for more information about changing libraries at link time.

2.5.1.3 The INCLUDE Variable

The **INCLUDE** environment variable defines the standard places where the compiler searches for include files.

An include file is a file incorporated into another source file with the **$INCLUDE** metacommand. The compiler searches the standard places for all files included in your program after searching the directory containing the source file that has the **$INCLUDE** metacommand. The **DEMOEXEC.FOR** demonstration program on the Utilities and Source Code distribution disk illustrates the use of include files. Otherwise, no include files are provided with the Microsoft FORTRAN Compiler; this feature is provided as a convenience for your own include files.

The **/I** and **/X** options, discussed in Section 3.3.8, let you temporarily change the search path for include files without affecting the **INCLUDE** variable. Section 3.3.8 also lists the places that the compiler searches for include files and the order in which these places are searched.

2.5.1.4 The TMP Variable

The compiler creates a number of temporary files as it processes a program. The **TMP** environment variable tells the compiler and the operating system where to create these files. The temporary files are removed by the time the compiler finishes processing.

The space required for the temporary files is typically double the size of the source file. It is often helpful to create the temporary files on a memory-based disk emulator, commonly referred to as a "RAM disk," as described in Section 2.7. You can speed processing by assigning the drive name you use for a RAM disk to the **TMP** variable.

2.5.1.5 Setting Environment Variables

Use the DOS **SET** command to assign a directory specification or specifications to the environment variables **PATH**, **INCLUDE**, **LIB**, and **TMP**. You must set **PATH**, **INCLUDE**, and **TMP** *before* invoking the compiler if you want the settings of these variables to be in effect while the compiler is running. Similarly, you must set **LIB** before you run the linker if you want the linker to use the new setting.

Whereas the **TMP** variable can be assigned only one path name, the **INCLUDE**, **PATH**, and **LIB** variables can each contain more than one path name. Each path name is separated from the next path name by a semicolon (;). The compiler or linker searches through all directories specified, in order of their appearance, until it finds the file it needs. This means that include files, executable files, and libraries can be separated and placed in different directories.

For example, you can tell the compiler where to look for include files by setting the **INCLUDE** variable, as shown below:

```
SET INCLUDE=C:\CUSTOM;C:\INCLUDE
```

First the compiler searches for include files in the directory containing the source file with the **$INCLUDE** metacommand. Next, it looks on Drive C in the subdirectory named \CUSTOM. Finally, if necessary, the compiler searches in the subdirectory named \INCLUDE on Drive C.

Important

Use the **PATH** command instead of the **SET** command to define the **PATH** variable. Although you can define the **PATH** variable with the **SET** command, using the **SET** command under versions of DOS earlier than 3.0 can cause the **PATH** variable to work incorrectly for some path specifications that use lowercase letters.

To define the **PATH** variable with the **PATH** command, simply give the **PATH** command followed by a space (or an equal sign) and one or more directory specifications separated by semicolons. For example, you might use the following command line:

```
PATH C:\BIN;C:\LINKER
```

This tells the compiler and the operating system to search for executable files on Drive C in the directory named \BIN, then, if necessary, in the \LINKER directory.

Important

The environment table, which holds any environment variables you have set and the values you have assigned, is 160 bytes by default. If you want to set up a complex environment, this may not be enough space. If you are running on IBM PC-DOS Version 3.1 or earlier, you can use the **SETENV** program to increase the size of the environment table. See Section 7.4 for more information.

Once you have set an environment variable, it remains effective until you reset it to a different value (or to an empty value) or until you turn off the machine. You can place the **SET** and **PATH** commands in your **AUTOEXEC.BAT** file after you run **SETUP**, so that you can automatically set these variables to the values you want when you boot your machine.

You can also use **SET** and **PATH** commands in a DOS batch file to define the environment for a particular program or programs. If you frequently switch between different environments, you can save time by setting up batch files that contain the **SET** and **PATH** commands for each environment. Then you can execute the batch file each time you want to switch to a new environment.

2.5.2 CONFIG.SYS Settings

After you run **SETUP**, you may have to create or modify settings in your **CONFIG.SYS** file, as described in this section, to allow the compiler to run.

First of all, you must have a **CONFIG.SYS** file. If you don't, use any text editor to create a file named **CONFIG.SYS** on your system disk (or root directory if you boot from your hard disk) and insert the lines described below in your **CONFIG.SYS** file.

The compiler must be able to open at least 15 files at one time. Check this by looking in your **CONFIG.SYS** file for the following line:

`files=`*number*

If *number* is less than 15, edit **CONFIG.SYS** to set *number* to an integer greater than 15. See the *Microsoft MS-DOS Programmer's Reference* for more information on this setting.

Note

If you do not specify enough files in the `files=` command in your **CONFIG.SYS** file, you may see one of the following fatal error messages during compilation:

```
cannot open compiler intermediate file - no more files
```

or

filename `: cannot open include file`

It is recommended, although not required, that you also set the number of buffers allowed in your **CONFIG.SYS** file. Check your **CONFIG.SYS** for the following line:

```
buffers=number
```

If *number* is not already set, 10 is a reasonable number. See your *Microsoft MS-DOS Programmer's Reference* for more information on this setting.

After you have edited or created your **CONFIG.SYS** file, reboot the system so the new settings take effect.

2.6 Using an 80186, 80188, or 80286 Processor

You can use the compiler with an 80186, 80188, or 80286 processor without taking any special steps. However, to take advantage of your processor's capabilities, you will probably want to use the **/G1** or **/G2** option when you compile your programs. These options enable the instruction set for the 80186/80188 and 80286 processors, respectively. See Section 3.3.14 for more information about these options.

2.7 Using a RAM Disk

If you have sufficient available memory, you can set up your memory to run portions of the compiler from a RAM disk. Using a RAM disk allows you to compile programs considerably faster than you could otherwise.

If you are using a RAM disk, you can set the value of the **TMP** environment variable to the drive name that you are using for the RAM disk. In this way, you can use the RAM disk for temporary files during compilation. Since temporary files are typically twice the size of the source file, you need approximately twice as much available memory as the size of the source file you are compiling.

Another way to use a RAM disk is to copy the library you will be using to the RAM disk with the DOS **COPY** command. This option requires at least 256K of available memory in addition to the 320K needed for the compiler and any additional memory needed for DOS and memory-resident programs. After you copy the library to the RAM disk, you must set the **LIB** environment variable to the drive name you are using for the RAM disk.

2.8 Converting Existing FORTRAN Programs

If you are upgrading from Version 3.2 or Version 3.3 of the Microsoft FORTRAN Compiler, see Appendix A for a discussion of differences between this compiler and earlier versions.

2.9 Quick Start

This section aims to help you quickly begin compiling and linking programs by giving examples of command lines and options. For a step-by-step approach to the compiling and linking process, see Section 2.10, "Practice Session."

The **FL** command lines given in the following sections illustrate some of the most common command-line options. You can use these command lines exactly as shown to get started with the compiler and linker, or you can use them as models and supply your own combination of options.

See Chapter 3 for an in-depth discussion of how the **FL** command line works. Chapter 4 fully describes the linker and its options.

Each option illustrated in this section is defined in full elsewhere in this manual. Use the index at the back of this manual to find more information on a particular option.

The **FL** command invokes both the compiler and the linker, so you don't need to give separate commands for compiling and linking (although you can). Notice that no library names are given at link time in the commands shown below. At compile time, the **FL** command automatically places in the object files the name of the standard FORTRAN library for the memory-model and floating-point options you have chosen. Thus, you don't have to give library names when you link, as long as the libraries you created use the default names for the **SETUP** options you have chosen.

2.9.1 Quick Overview

For a quick overview of commonly used compiler options, type the following at the DOS prompt:

```
FL /HELP
```

The **/HELP** option displays a categorized summary of **FL** options.

The *Microsoft FORTRAN Compiler Quick Reference Guide* that accompanies this manual is another good source for a quick overview. It lists the **FL** options in alphabetical order.

2.9.2 Simple Compile and Link

```
FL FILE1.FOR FILE2.FOR
```

The example above demonstrates compiling and linking two files named FILE1.FOR and FILE2.FOR. Two object files, FILE1.OBJ and FILE2.OBJ, are created. Since no memory-model or floating-point options are given, these object files are linked with the appropriate library for the default memory model (large) and the floating-point math package (8087/80287), **LLIBFOR7.LIB**. The executable file is named FILE1.EXE.

2.9.3 Using Wild Cards

```
FL /FePROGRAM /Fs *.FOR
```

The command above compiles and links all source files in the current working directory. The **/Fe** option gives the resulting executable file the name PROGRAM. The **/Fs** option creates a source-listing file for each source file; each source-listing file has the same base name as the corresponding source file, but has the extension **.LST** instead of **.FOR**. (The base name of a file is the portion of the name preceding the period.)

2.9.4 Compiling without Linking

```
FL /c FILE.FOR
```

The command above compiles but does not link the given file. You can also use the **FL** command to link without compiling by just giving object files on the command line. For example,

```
FL FILE.OBJ
```

invokes the linker to create an executable program named `FILE.EXE`.

2.9.5 Using the Emulator Library

```
FL /FPc EMULAT.FOR
```

By default, Microsoft FORTRAN programs handle floating-point operations by generating in-line instructions for an 8087 or 80287 math coprocessor and using that coprocessor for floating-point math. The command shown above creates a program that handles floating-point math differently: the program generates calls to floating-point functions in an emulator library. The emulator library emulates most of the 8087/80287 functions in software.

This program uses a coprocessor if one is present. In addition, this program can be linked with a library other than the emulator library. See Section 8.3 for a detailed description of floating-point options and their effects.

2.9.6 Preparing to Use the CodeView Debugger

```
FL /Zi /Od FILE.FOR
```

The example above uses the **/Zi** option to create object and executable files that contain symbol-table information for debugging with the Microsoft CodeView window-oriented debugger. It also uses the **/Od** option to disable optimization and make it easier to debug and correct code. See Section 3.3.13, "Preparing for Debugging," for more information.

2.9.7 Using the Debug and Declare Options

```
FL /Zi /4Ybd FILE.FOR
```

The example above enables extended error handling at run time (as if a **$DEBUG** metacommand appeared at the top of the source file) and causes the compiler to generate warnings about undeclared variables (as if a **$DECLARE** metacommand appeared at the top of the source file). The **/Zi** option, which prepares the file for use with the Microsoft CodeView debugger, is also used. See Section 3.3.9.2 for more information about the debug and declare options.

2.9.8 Setting Titles and Subtitles

```
FL /Fs /St'Main Title' /Ss'Subtitle' /Sp20 /Sl90 FILE.FOR
```

The example above compiles and links FILE.FOR, creating an executable file named FILE.EXE. The **/Fs** option creates a source-listing file named FILE.LST. The listing has a main title and subtitle; it is 20 lines long and 90 characters wide. See Sections 3.3.7.1, "Types of Listings," and 3.3.7.4, "Titles and Subtitles," for more information.

2.9.9 Compiling a Free-Form File

```
FL /c /4Yf FILE.FOR
```

The example above compiles **FILE.FOR** in free-form format, creating an object file named FILE.OBJ. See Section 3.3.11, "Controlling Source-File Syntax," for more information about free-form files.

2.10 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft FORTRAN Compiler. By following these steps you can produce and run an executable program file.

Note

This practice session gives you a step-by-step introduction to the basic form of the **FL** command. Also see Section 2.9, "Quick Start," which shows sample command lines and options that illustrate common compiling and linking operations.

The source file used for this practice session is the sample source file **DEMO.FOR**, which is included on your Utilities and Source Code distribution disk. **DEMO.FOR** is a simple FORTRAN program that performs a bubble sort on 10 values.

This practice session assumes that you have used the **SETUP** program to install the software, that you have set up the compiler environment as described in Section 2.5, and that you have copied **DEMO.FOR** to the directory or disk where you want to do your compiling and linking.

Once you have set up the environment, you are ready to begin processing **DEMO.FOR** using the following procedure:

1. Make sure that the directory containing **DEMO.FOR** is your current working directory (use the DOS **CD** command to change directories, if necessary).

Important

If you are using a floppy-disk system, insert the disk containing **DEMO.FOR** in Drive B; the Driver or Driver/Compiler disk containing the compiler passes should be in Drive A. Drive B should be the current drive. If you have changed your environment so that you can reverse drives, insert the disk containing **DEMO.FOR** in Drive A and the Driver or Driver/Compiler disk in Drive B, and make Drive A the current drive.

---

2. If you have installed an 8087/80287 library (the default library), type

```
FL /Fs DEMO.FOR
```

If you have installed an emulator library because you do not have a coprocessor, type

```
FL /FPi /Fs DEMO.FOR
```

First, the **FL** command invokes the compiler, which prints a message similar to the following message on your screen and begins to compile the source file:

```
Microsoft (R) FORTRAN Optimizing Compiler Version 4.00
Copyright (C) Microsoft Corp 1987.  All rights reserved.
```

The **/Fs** option creates a source listing named **DEMO.LST** in the current working directory.

Important

If you are using a floppy-disk system, the **FL** program prompts you when you need to insert the disk containing the next compiler pass or the linker.

---

3. The next message you see is similar to the following:

```
Microsoft (R) Overlay Linker Version 3.54
Copyright (C) Microsoft Corp 1984, 1985, 1986. All rights reserved.
```

This means that compilation is completed and the file is now being linked to form an executable program.

4. When the linking process is finished, the DOS prompt reappears. Your current working directory now has an executable file named **DEMO.EXE**. It also contains an object file named **DEMO.OBJ** and a source-listing file named **DEMO.LST**.

You may want to examine the source-listing file to familiarize yourself with its format. However, the source-listing file is not required for running the program, so you can delete it if you like.

You can also delete the object file (**DEMO.OBJ**); since you have the executable program file, it is no longer needed. See Chapter 5 for a discussion of how to use the Microsoft Library Manager, **LIB**, to organize object files into libraries of useful functions.

5. Type

   ```
   DEMO
   ```

   to run the sample program.

2.11 Using Batch Files

You can create a DOS batch file to set up the compiler environment and execute the **FL** command. Batch files are useful with the **FL** command because they allow you to set up an environment before using the command. Creating and using batch files is discussed more fully in your DOS manual; this section is intended only to demonstrate a few of the possible uses of the **FL** command in a batch file.

Note

The **MAKE** utility, discussed in Chapter 6, "Maintaining Programs with MAKE," is a more sophisticated tool than a batch file for automating the compiling and linking process, especially for a large set of software. You may want to try setting up a **MAKE** file instead of using a batch file.

The following batch file, `MYCOMP.BAT`, could be used to create a program and a map file (described in Section 3.3.7.5, "Formats for Listings") from a FORTRAN source file in an environment set up for that purpose:

```
SET INCLUDE=C:\TOP\MYINC
FL /c %1.FOR
IF NOT ERRORLEVEL 1 LINK %1,,%1;
```

The value given to **INCLUDE** in the first line alters the environment for the **FL** command. Since no value is given for **PATH**, **TMP**, or **LIB**, their current values, if set, are unaffected by the batch file.

To run the batch file, type the following line:

```
MYCOMP THIS
```

The file name THIS is substituted for %1, and THIS.FOR is compiled, producing the object file THIS.OBJ. The **/c** option means that the file is compiled but not linked.

The second line of the batch file ensures that linking is attempted only if the source file was successfully compiled. The **FL** command returns an exit code to allow testing for successful compilation. The exit code 0 indicates success (see Appendix B for information on other exit codes). The DOS batch command **IF ERRORLEVEL** is used to test whether the exit code is 1 or greater. See your DOS documentation for more information on this command.

If compilation is successful, the object file THIS.OBJ is linked to produce THIS.EXE (the default name, since none is supplied).

Note that the value given to **INCLUDE** when you execute the batch file remains in effect until you explicitly change it or until you reboot your machine. To restore your usual environment settings, you can create a batch file that resets the environment variables to the directories you most frequently use. For example, the following lines might be placed in a file called RESET.BAT, to be executed by typing RESET whenever you want to restore your usual environment settings:

```
PATH C:\BIN
SET INCLUDE=C:\INCLUDE
SET LIB=C:\LIB
SET TMP=C:\
```

Compiling and Linking

Chapter 3
Compiling: The FL Command

3.1 Introduction 51

3.2 The Basics: Compiling, Linking, and Running FORTRAN Files 52

3.2.1 Compiling and Linking with FL 52

3.2.1.1 Stopping FL 54

3.2.1.2 Swapping Disks 54

3.2.2 Linking with Libraries 55

3.2.3 Running Your FORTRAN Program 56

3.3 Using FL Options 58

3.3.1 Floating-Point (/FL) and Memory-Model (/A) Options 59

3.3.2 Getting Help with FL Options (/HELP) 61

3.3.3 Specifying Source Files (/Tf) 62

3.3.4 Compiling without Linking (/c) 63

3.3.5 Naming the Object File (/Fo) 64

3.3.6 Naming the Executable File (/Fe) 65

3.3.7 Creating Listing Files 66

3.3.7.1 Types of Listings (/F Options) 67

3.3.7.2 Special File Names 70

3.3.7.3 Line Size (/Sl) and Page Size (/Sp) 70

3.3.7.4 Titles (/St) and Subtitles (/Ss) 72

3.3.7.5 Formats for Listings 73

3.3.8 Searching for Include Files (/I, /X) 81

3.3.9 Handling Warnings and Errors 84
3.3.9.1 Understanding Error Messages 85
3.3.9.2 The Debug (/4Yb, /4Nb) and Declare (/4Yd, /4Nd) Options 86
3.3.9.3 The Warning-Level Option (/W) 89
3.3.9.4 Syntax Errors (/Zs) 90
3.3.10 Setting the Default Integer Size (/4I2, /4I4) 90
3.3.11 Controlling Source-File Syntax (/4Y6, /4N6, /4Yf, /4Nf, /4Ys, /4Ns, /4Yt, /4Nt) 91
3.3.12 Conditional Compilation (/4cc) 93
3.3.13 Preparing for Debugging (/Zi, /Od, /Zd) 94
3.3.14 Using an 80186, 80188, or 80286 Processor (/G0, /G1, /G2) 96
3.3.15 Optimizing (/O Options) 97
3.3.16 Enabling Stack Probes (/Ge) 100
3.3.17 Suppressing Automatic Library Selection (/Zl) 101
3.3.18 Setting the Stack Size (/F) 102
3.3.19 Restricting the Length of External Names (/H) 103
3.3.20 Labeling the Object File (/V) 103
3.3.21 Compatibility with Version 3.2 (/Gr) 104
3.4 Using FL to Link without Compiling 104

3.1 Introduction

This chapter explains how to compile and link using the **FL** command and discusses commonly used **FL** options. The **FL** command is the only command you need to compile and link your FORTRAN source files. **FL** executes the three compiler passes, then automatically invokes **LINK**, the Microsoft Overlay Linker, to link your files.

Using the **FL** options described in this chapter, you can control and modify the tasks performed by the command. For example, you can direct **FL** to create listings of your source file or the object code created by the compiler.

Other **FL** options let you give information for the compilation process. Two important options specify how the program being compiled will handle floating-point operations and which memory model the program will use. Others give the search path for files to be included, provide titles and subtitles for source-code listings, and specify whether or not the program can generate compiler warning messages.

The **FL** command automatically optimizes your program for fast execution time. You never need to give an optimizing option, unless you want to change the way **FL** optimizes or you want to disable optimization altogether. See Section 3.3.15, "Optimizing," for more information on these choices.

Note

This chapter assumes that you know how to create, edit, and debug FORTRAN program files on your system. For questions relating to the definition of the FORTRAN language, see the *Microsoft FORTRAN Compiler Language Reference*. The Microsoft CodeView manual explains how to use the symbolic debugger provided with this package.

For a quick introduction to running the compiler and linker using **FL**, see Section 2.9, "Quick Start."

Section 3.2 explains the basic use of the **FL** command to produce an executable program. It also describes how to run the program and pass command-line arguments to the program, if desired.

Sections 3.3.1 – 3.3.21 introduce commonly used **FL** options. The **FL** options that control floating-point operations and memory models are discussed in Chapters 8 and 9, respectively. A summary of the **FL** command and all available options appears in the *Microsoft FORTRAN Compiler Quick Reference Guide* provided with this package.

3.2 The Basics: Compiling, Linking, and Running FORTRAN Files

This section explains how to use **FL** to compile and link FORTRAN files, and discusses the rules and conventions that apply to the file names and options used with **FL**. It also explains how to run the executable program created by **FL**.

3.2.1 Compiling and Linking with FL

The **FL** command has the following form:

FL [[*option...*]] [[*filespec...*]] [[*option...*]] [[*filespec...*]]
[[**/link** [[*libfield*]] [[*linkoptions*]]]]

Note

Syntax that is too long to fit on one line is continued on two or more lines.

Each *option* is one of the command-line options described in this manual, and each *filespec* names a file to be processed. The **FL** command automatically specifies the appropriate library to be used during linking; however, you can use the **/link** option with the optional *libfield* and *linkoptions* to specify additional libraries and options to be used during linking. See Section 3.4, "Using FL to Link without Compiling," for more information.

You can give any number of options and file names on the command line, provided that the command line does not exceed 128 characters. See Section 3.3 for more information on the rules governing the use of command-line options with **FL**.

The **FL** command can process source files, object files, or a combination of source and object files. It uses the file-name extension (the period plus any letters that follow it) to determine what kind of processing the file needs, as shown below:

- If the file has a **.FOR** extension, **FL** compiles the file.
- If the file has an **.OBJ** extension, **FL** processes the file by invoking the linker. (Linking with the **FL** command is discussed in Section 3.4.)
- If the extension is omitted or is anything other than **.FOR** or **.OBJ**, **FL** assumes the file is an object file unless the file name appears in a **/Tf** option. If the file name appears in a **/Tf** option, **FL** assumes the file is a FORTRAN source file. See Section 3.3.3 for a description of the **/Tf** option.

You can use the DOS wild-card characters (* and ?) to process all files meeting the wild-card specification, as long as the files have the required extensions. See the DOS manual for more information on wild-card characters.

Any *filespec* on the **FL** command line can include a full or partial path specification, allowing you to process files in different directories or on different drives. A full path specification starts with the drive name; a partial path specification gives one or more directory names before the name of the file, but does not give a drive name.

You can use uppercase letters, lowercase letters, or a combination of both for the file names on the **FL** command line. For example, the following three file names are equivalent:

```
abcde.for
ABCDE.FOR
aBcDe.for
```

When **FL** compiles source files, it creates object files. By default, these object files have the same base names as the corresponding source files, but with the extension **.OBJ** instead of **.FOR**. (The base name of a file extension is the portion of the name preceding the period, but excluding the path specification and drive name, if any.) You can use the **/Fo** option to assign a different name to an object file.

These object files, along with any **.OBJ** files given on the command line, are linked to form an executable program file. The executable file has the base name of the first file (source or object) on the command line, plus an **.EXE** extension. If only **.OBJ** files are given on the command line, the compilation stage is skipped altogether, and the files are simply linked.

You can tell whether **FL** is compiling or linking by the messages that appear on the screen. When **FL** invokes the compiler, a message similar to the following message appears on your screen:

```
Microsoft (R) FORTRAN Optimizing Compiler Version 4.00
Copyright (C) Microsoft Corp 1987.  All rights reserved.
```

As each source file on the command line is compiled, its name appears on the screen. When all source files have been compiled and the linker is invoked, a message similar to the following message appears:

```
Microsoft (R) Overlay Linker  Version 3.54
Copyright (C) Microsoft Corp 1984, 1985, 1986. All rights reserved.
```

This message is followed by several lines showing Microsoft **LINK** prompts and the responses provided by **FL**. The **FL** command uses the response-file method of invoking Microsoft **LINK**. See Chapter 4, "Linking," for more information on how the **LINK** prompts and responses work.

Note

FL always uses the **/NOI** (**NOIGNORECASE**) link option in the linking stage. This means, for example, that the linker regards `Global` and `GLOBAL` as two different symbols. If you want the linker to ignore, rather than consider case, you must link in a separate step using the **LINK** command without the **/NOI** option. See Section 4.6.7 for more information.

3.2.1.1 Stopping FL

If you want to stop the compiling and linking session for any reason, press CONTROL-C or CONTROL-BREAK. You will be returned to the DOS command level, where you can restart **FL**.

3.2.1.2 Swapping Disks

If you are running the compiler on a floppy-disk system, you will need to swap disks during the compilation process. **FL** suspends execution and prompts you to insert a disk and press any key. After you insert the disk and press a key as directed, compilation and linking continue.

- **Examples**

```
FL A.FOR B.FOR C.OBJ D
```

The command line above compiles the files A.FOR and B.FOR, creating object files named A.OBJ and B.OBJ. These object files are then linked with C.OBJ and D.OBJ to form an executable file named A.EXE (since the base name of the first file on the command line is A.) Note that the extension .OBJ is assumed for D since no extension is given on the command line.

```
FL *.FOR
```

The command line above compiles all source files with the default extension (**.FOR**) in the current working directory. The resulting object files are linked to form an executable file whose base name is the same as the base name of the first file compiled.

```
FL *.OBJ
```

The command above links all object files with the default extension (**.OBJ**) in the current working directory, creating an executable file whose base name is the same as the base name of the first object file.

3.2.2 Linking with Libraries

When the **FL** command compiles a source file, it places the name of a FORTRAN library in the object file that it creates. The library name corresponds to the memory-model and floating-point options that you chose on the **FL** command line, or the defaults for options you did not explicitly choose. See Table 3.1 in Section 3.3.1, "Floating-Point and Memory-Model Options," for the library names that **FL** includes in the object file for each combination of memory-model and floating-point options.

When you link the object file, the linker looks for a library matching the name embedded in the object file. If it finds a library matching this name, it automatically links the library with the object file.

The result is that you do not need to give library names on the **FL** command line unless you want to link with libraries that you have renamed during **SETUP** to names other than the "standard" names (see Section 2.4.6.3, "Naming Libraries") or with libraries other than the appropriate library for the floating-point and memory-model options you have chosen.

If you want to link with other libraries, you must either give the **/link** option on the **FL** command line and include the new library names, or run the linker and specify the library names separately. In either case, the linker searches the library you specified to resolve external references before it searches the library whose name is embedded in the object file. If you want the linker to ignore the library whose name is embedded in the object file, you must also include the **/NOD** linker option, either as part of the **/link** option on the **FL** command line or as an option on the **LINK** command line.

See Section 3.4, "Using FL to Link without Compiling," for information about the **/link** option of the **FL** command. See Section 4.5.3, "Specifying Libraries to Be Searched," for information about specifying library names to the linker.

3.2.3 Running Your FORTRAN Program

Once you have compiled and linked a program, you have an executable file with the extension **.EXE**. This file can be run from the DOS prompt. DOS uses the **PATH** environment variable to find executable files. You can execute your program from any directory, as long as the executable program file is either in your current working directory or in one of the directories given by the **PATH** variable.

When you run your program, you can give file names on the same command line you use to execute the program.

The file names on the command line are used to satisfy **OPEN** statements in your program that leave the file-name field blank. The first file name on the command line is used for the first such **OPEN** statement executed, the second file name is used for the second statement, and so on. (See Section 5.3.38 of the *Microsoft FORTRAN Compiler Language Reference* for a description of the **OPEN** statement.)

Note

If your program executes a **READ** or **WRITE** statement specifying a file that has not been opened, the effect is the same as that of an **OPEN** statement with a blank file name. Default values are assigned to the parameters normally given in the **OPEN** statement.

If the file names on the command line outnumber the **OPEN** statements with blank file names, the extra file names are ignored.

If more **OPEN** statements with blank file names are executed than there are file names on the command line, you will be prompted to enter a file name for each additional **OPEN** statement. You are also prompted if you give a null file name; see the example below.

Each file name on the command line must be separated from the names around it by one or more spaces or tab characters. The file name can be enclosed in double quotation marks (" ") if desired, but this is not required. A null argument consists of just an empty set of double quotation marks, with no file name enclosed.

■ Example

```
MYPROG "" OUTPUT.DAT
```

This example runs the program MYPROG.EXE. Since the first file-name argument is null, the first **OPEN** statement with a blank file-name field produces the following message:

```
File name missing or blank -
Please enter name UNIT number
```

The *number* is the unit number specified in the **OPEN** statement. The file name OUTPUT.DAT is used for the second such **OPEN** statement executed. If additional **OPEN** statements with blank file-name fields are executed, you will be prompted for additional file names.

3.3 Using FL Options

The **FL** command offers a large number of command-line options to control and modify the compiler's operation. Options begin with the forward slash character (**/**) and contain one or more letters. You can use a dash (**-**) instead of the forward slash if you prefer. For example, **/I** and **-I** are both acceptable forms of the **I** option. In this manual, forward slashes are used for options, although in error messages, dashes are used.

Important

Although file names can be given in either uppercase or lowercase, options must be given exactly as shown. For example, **/Zd** is a valid option, but **/ZD** and **/zd** are not.

Options can appear anywhere on the **FL** command line. In general, an option applies to all files following it on the command line, and does not affect files preceding it on the command line. However, not all options follow this rule; see the discussion of a particular option for information on its behavior. Keep in mind that most **FL** options apply only to the compilation process. Unless specifically noted, options do not affect any object files given on the command line.

Some options take arguments, such as file names, strings, or numbers. Spaces are not allowed between the option letter and the argument. For example, the numerical argument to the **/Sp** option must be given as shown below:

```
/Sp60
```

Some options consist of more than one letter. For example, the **/Sp** option shown above is a two-letter option. No spaces are allowed between the letters of an option. Thus, `/S p60` would cause a command-line error.

Sections 3.3.1 – 3.3.21 discuss the basic **FL** options and the tasks they perform. Chapters 8 and 9 describe **FL** options for floating-point operations and memory models, respectively. Additional linking options are discussed in Sections 4.6.1 – 4.6.17.

3.3.1 Floating-Point (/FL) and Memory-Model (/A) Options

Two important options that you specify with the **FL** command are the following:

1. How your program handles floating-point operations
2. The memory model used for your program

The **FL** command includes the following options that allow you to choose how the program you are compiling will handle floating-point operations:

Option	Effect
/FPi87	Generates in-line instructions and selects the 8087/80287 math package
/FPi	Generates in-line instructions and selects the emulator math package
/FPc87	Generates floating-point calls and selects the 8087/80287 math package
/FPc	Generates floating-point calls and selects the emulator math package
/FPa	Generates floating-point calls and selects the alternate math package.

See Chapter 8 for a description of these options and their effects.

You use the **FL** command to specify the memory model your program will use. The memory model defines the rules that the compiler will use to set up the program's code and data segments in memory. **FL** offers the following memory-model options:

Option	Effect
/AL	Chooses the large memory model (default)
/AM	Chooses the medium memory model
/AH	Chooses the huge memory model

See Chapter 9 for a description of these options and the memory models they specify.

The floating-point and memory-model options you choose determine the name of the standard library that **FL** places in the object file it creates. This library is then considered the default library, since the linker searches for it by default. Table 3.1 shows each combination of memory-model and floating-point options and the corresponding library name that **FL** embeds in the object file.

Table 3.1

FL Options and Default Libraries

Memory-Model Option	Floating-Point Option	Default Library
/FPi87 or **/FPc87**	**/AL** or **/AH**	**LLIBFOR7.LIB**
	/AM	**MLIBFOR7.LIB**
/FPi or **/FPc**	**/AL** or **/AH**	**LLIBFORE.LIB**
	/AM	**MLIBFORE.LIB**
/FPa	**/AL** or **/AH**	**LLIBFORA.LIB**
	/AM	**MLIBFORA.LIB**

Note

If you have renamed any of the libraries you created while running **SETUP**, the library name embedded in the object file might not match the renamed library. In these cases, you must explicitly specify the new library name to the linker. See Section 3.2.2, "Linking with Libraries," for more information.

3.3.2 Getting Help with FL Options (/HELP)

■ Options

/HELP
/help

The **/HELP** option displays a list of the most commonly used FORTRAN options. (See the *Microsoft FORTRAN Compiler Quick Reference Guide* for a complete alphabetical list of **FL** options.) For this option to work, the file containing the FORTRAN options, **FL.HLP**, must be in the current directory or in the path given in the **PATH** environment variable. If **FL** cannot find this file, it displays the following error message:

```
cannot open help file, 'fl.hlp'
```

When the **/HELP** option appears on the **FL** command line, **FL** displays the list of options but does not take any other action, regardless of what other information appears on the command line. For example, if you give a source-file name along with the **/HELP** option, **FL** does not compile the source file.

This option is not case sensitive. Any combination of uppercase and lowercase letters will work.

The help screen prompts you to press any key before returning to the DOS prompt. This keeps the top lines of the help screen in view; once you press the key and return to the DOS prompt, the top lines scroll out of view.

■ Examples

The following examples show how you can save the help screen for future reference by sending it to a file or printer:

```
FL /HELP > HELP.DOC
```

The example above saves the help screen in a file named HELP.DOC.

```
FL /HELP >PRN
```

The example above sends the screen output directly to the printer device, PRN. (See Section 3.3.7.2, "Special File Names," or your DOS documentation for a list of device names that can be used in redirection.)

Note that you may have to press the ENTER key several times to make sure that all of the help messages are saved or printed. Since the messages may be displayed on several separate screens, **FL** waits for you to enter a keystroke before displaying the next screenful of messages. Also, you must press an additional key (any key can be used, including the ENTER key) after giving the **FL** command, since the help screen requires you to press a key before returning to the DOS prompt.

3.3.3 Specifying Source Files (/Tf)

■ Option

/Tf[ ][*sourcefile*

The **/Tf** option tells the **FL** command that the given file is a FORTRAN source file. If this option does not appear, **FL** assumes that files with the extension **.FOR** are FORTRAN source files, and files with any other extension or with no extension are object files. If you use the **/Tf** option, **FL** treats the given file as a FORTRAN source file, regardless of its extension. A separate **/Tf** option must appear for each source file that has an extension other than **.FOR**. The space between **/Tf** and *sourcefile* is optional.

If you have to specify more than one source file with an extension other than **.FOR**, it is safest to give each source file in a separate **/Tf** option. Although a *sourcefile* with a wild-card character is legal, this use of wild-card characters may cause problems. If a *sourcefile* with a wild-card character represents a single file, then **FL** behaves as expected: it considers that single file to be a FORTRAN source file. However, if a *sourcefile* with a wild-card character represents more than one file, **FL** treats only the first file as a FORTRAN source file. It treats any other files that the *sourcefile* represents as object files.

■ Examples

```
FL MAIN.FOR /TfTEST.PRG /TfCOLLATE.PRG PRINT.PRG
```

In the example above, the **FL** command compiles the three source files MAIN.FOR, TEST.PRG, and COLLATE.PRG. Since the file PRINT.PRG is given without a **/Tf** option, **FL** treats it as an object file. Thus, after compiling the three source files, **FL** links the object files MAIN.OBJ, TEST.OBJ, COLLATE.OBJ, and PRINT.PRG.

```
FL /TfTEST?.F00
```

Assume that the **FL** command in the example above is entered when the files TEST1.F00, TEST2.F00, and TEST3.F00 all exist in the current directory. The **FL** command in this example would compile TEST1.F00 as a FORTRAN program and then try to treat TEST2.F00 and TEST3.F00 as object files. The **FL** command shown above would have the same effect as the following **FL** command:

```
FL TEST1.FOR TEST2.F00 TEST3.F00
```

3.3.4 Compiling without Linking (/c)

- **Option**

/c

The **/c** (for "Compile-only") option suppresses linking. Source files given on the command line are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified on the command line are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The **/c** option applies to the entire **FL** command line, regardless of the option's position in the command line.

- **Example**

```
FL /c *.FOR
```

This command line compiles, but does not link, all files with the extension **.FOR** in the current working directory.

3.3.5 Naming the Object File (/Fo)

- **Option**

/Fo*objfile*

By default, **FL** gives each object file the same base name as the corresponding source file, plus the extension **.OBJ**. The **/Fo** option lets you give an object file a different name or create it in a different directory.

The *objfile* argument must appear immediately after the option, with no intervening spaces. The *objfile* argument can be a file specification, a drive name, or a path specification.

If *objfile* is a file specification, the **/Fo** option applies only to the source file immediately following the option on the command line. The object file created by compiling that source file has the name given by *objfile*.

If *objfile* is a drive name or path specification, the **FL** command creates object files in the given location for every source file following the **/Fo** option on the command line. The default names are used for the object files; that is, each object file has the base name of the corresponding source file, with the **.OBJ** extension replacing the **.FOR** extension.

Important

When you give just a path specification, the *objfile* argument must end with a backslash (\) so that **FL** can distinguish between it and an ordinary file name.

You may supply any name and any extension you like for *objfile*. However, it is recommended that you use the conventional **.OBJ** extension because the **FL** command, as well as the **LINK** and **LIB** utilities, expects the **.OBJ** extension when processing object files. If you give an object-file name that lacks an extension, **FL** automatically appends the **.OBJ** extension. However, if you give an object-file name with a *blank* extension (that is, an object-file name that ends with a period), **FL** does not append an extension.

■ Examples

```
FL /c /FoSUB\THAT THIS.FOR
```

The example above compiles the file THIS.FOR and creates an object file named THAT.OBJ in the subdirectory SUB. Note that **FL** automatically appends the **.OBJ** extension. Linking is suppressed because the **/c** option is given.

```
FL /FoB:\OBJECT\ *.FOR
```

The example above compiles and links all source files with the extension **.FOR** in the current working directory. The option /FoB:\OBJECT\ tells **FL** to create all the object files in the directory named OBJECT on drive B. Each object file has the base name of the corresponding source file, plus the extension **.OBJ**.

3.3.6 Naming the Executable File (/Fe)

■ Option

/Fe*exefile*

By default, the executable file produced by the **FL** command is given the base name of the first file (source or object) on the command line, plus the extension **.EXE**. The **/Fe** option lets you give the executable file a different name or create it in a different directory.

Since only one executable file is created, it does not matter where the **/Fe** option appears on the command line. If more than one **/Fe** option appears, the last name on the command line prevails.

/Fe applies only in the linking stage; if **/c** is used to suppress linking, **/Fe** has no effect.

The *exefile* argument must appear immediately after the option, with no intervening spaces. The *exefile* argument can be a file specification, a drive name, or a path specification.

If *exefile* is a file specification, the executable file is given the specified name. If *exefile* is a drive name or path specification, the **FL** command creates the executable file in the given location, using the default name (base name of the first file plus **.EXE**).

Important

When you give a path specification as the *exefile* argument, the path specification must end with a backslash (\) so that **FL** can distinguish it from an ordinary file name.

You are free to supply any name and any extension you like for the *exefile*. If you give a file name without an extension, **FL** automatically appends the **.EXE** extension.

■ Examples

```
FL /FeC:\BIN\PROCESS *.FOR
```

The example above compiles and links all source files with the extension **.FOR** in the current working directory. The resulting executable file is named `PROCESS.EXE` and is created in the directory `C:\BIN`.

```
FL /FeC:\BIN\ *.FOR
```

The example above is similar to the first example except that the executable file, instead of being named `PROCESS.EXE`, is given the same base name as the first file compiled. The executable file is created in the directory `C:\BIN`.

3.3.7 Creating Listing Files

A number of listing options are available with the **FL** command. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set the title and subtitle of the source listing from the command line and control the length of source-listing lines and pages.

The **FL** command optimizes by default, so object listings reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the **/Fc** option (described below) to mingle the source and assembly code. To suppress optimization, use the **/Od** option, discussed in Section 3.3.15.

The options available for producing listings and controlling their appearance are described in the following sections.

Note

Listings produced by **FL** may contain names that begin with one or more underscores (for example, `__chkstk`) or that end with the suffix **QQ**. Names that use these conventions are reserved for internal use by the compiler.

3.3.7.1 Types of Listings (/F Options)

- **Options**

/Fs[*listfile*]	Produces source listing
/Fl[*listfile*]	Produces object listing
/Fa[*listfile*]	Produces assembly listing
/Fc[*listfile*]	Produces combined source and object listing
/Fm[*mapfile*]	Produces map file that lists segments, in order

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see Section 3.3.7.5, "Formats for Listings."

When using the options described in this section, the *listfile* argument, if given, must follow the option immediately, with no intervening spaces. The *listfile* can be a file specification, a drive name, or a path specification. It can also be omitted.

Important

When you give just a path specification as the *listfile* argument, the path specification must end with a backslash (\) so that **FL** can distinguish it from an ordinary file name.

When you give a drive name or path specification as the argument to a listing option, or if you omit the argument altogether, **FL** uses the default file name for the listing type. Table 3.2 gives the default names used for each type of listing. The table also shows the default extensions, which are used when you give a file-name argument that lacks an extension.

Table 3.2

Default File Names and Extensions

Option	Listing Type	Default File Name[1]	Default Extension[2]
/Fs	Source	Base name of source file plus **.LST**	**.LST**
/Fl	Object	Base name of source file plus **.COD**	**.COD**
/Fa	Assembly	Base name of source file plus **.ASM**	**.ASM**
/Fc	Combined source-object	Base name of source file plus **.COD**	**.COD**
/Fm	Map	Base name of first source or object file on the command line plus **.MAP**	**.MAP**

[1] The default file name is used when the option is given with no argument or with a drive name or path specification as the argument.

[2] The default extension is used when a file name lacking an extension is given.

Since you can process more than one file at a time with the **FL** command, the order in which you give listing options and the kind of argument you give for each option (file specification, path specification, or drive name) affect the result. Table 3.3 summarizes the effects of each option with each type of argument.

Table 3.3

Arguments to Listing Options

Option	File-Name Argument	Drive-Name or Path Argument[1]	No Argument
/Fa, **/Fc**, **/Fl**, **/Fs**	Creates a listing for next source file on command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
/Fm	Uses given file name for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name

[1] When you give just a path specification as the argument, the path specification must end with a backslash (\) so that **FL** can distinguish it from an ordinary file name.

If a source file includes one or more **$NOLIST** metacommands, the portion of the source file between each **$NOLIST** metacommand and the following **$LIST** metacommand (if any) is omitted from the listing.

Only one variation of the object or assembly listing can be produced for each source file. The **/Fc** option overrides the **/Fa** and **/Fl** options; whenever you use **/Fc**, a combined listing is produced. If you apply both the **/Fa** and the **/Fl** options to one source file, only the last listing specified is produced.

The map file is produced during the linking stage. If linking is suppressed with the **/c** option, the **/Fm** option has no effect.

3.3.7.2 Special File Names

You can use the DOS device names listed below as file-name arguments to the listing options. These special names allow you to direct listing files to your terminal or printer.

Name	Device
AUX	Refers to an auxiliary device.
CON	Refers to the console (terminal).
PRN	Refers to the printer device.
NUL	Specifies a "null" (nonexistent) file. Giving **NUL** as a file name means that no file is created.

Even if you add device designations or file-name extensions to these special file names, they remain associated with the devices listed above. For example, `A:CON.XXX` still refers to the console and is not the name of a disk file.

Important

When using device names, do not append a colon. The Microsoft FORTRAN Compiler does not recognize the colon. For example, use `CON` or `PRN`, not `CON:` or `PRN:`.

3.3.7.3 Line Size (/Sl) and Page Size (/Sp)

■ Options

/Sl[ ]*linesize*
/Sp[ ]*pagesize*

The default line size for source listings is 80 columns, and the page size is 63 lines. The **/Sl** and **/Sp** options let you change the line size and page size, respectively, for source listings. These options are useful for preparing source listings that will be printed on a printer that uses nonstandard page sizes. They are valid only if you also specify the **/Fs** option on the **FL** command line. The space between **/Sl** and *linesize*, or **/Sp** and *pagesize*, is optional.

The *linesize* argument gives the width of the listing line in columns (on line printers, columns usually correspond to characters). The number given must be a positive integer between 80 and 132, inclusive; any number that is outside this range produces an error message. Any line that exceeds the listing width is truncated.

The *pagesize* argument gives the number of lines to appear on each page of the listing. The minimum number is 15; if a smaller number is given, an error message appears and the default page size is used.

The **/Sl** or **/Sp** option applies to the remainder of the command line or until the next occurrence of **/Sl** or **/Sp** on the command line. These options do not cause source listings to be created. They take effect only when the **/Fs** option is also given to create a source listing.

You can use metacommands in the source file to override the **/Sl** and **/Sp** options. These options have the same effects as **$LINESIZE** and metacommands at the top of each file being compiled. If additional **$LINESIZE** or **$PAGESIZE** metacommands appear in the file being compiled, the line size or page size for that file is changed accordingly.

The **$LINESIZE** or **$PAGESIZE** metacommands in a particular file affect only that file and do not change the effects of **/Sl** or **/Sp** on any other files on the command line.

■ Examples

```
FL /c /Fs /Sl90 /Sp70 *.FOR
```

The example above compiles all source files with the default extension (**.FOR**) in the current working directory, creating a source-listing file for each source file. Each page of the source listing is 90 columns wide and 70 lines long.

```
FL /Fs /Sp70 MAIN.FOR /Sp63 SUB1.FOR SUB2.FOR
```

The example above compiles and links three source files, creating an executable file named MAIN.EXE. Three source listings are created: each page of MAIN.LST is 70 lines long, while each page of SUB1.LST and SUB2.LST is 63 lines long.

3.3.7.4 Titles (/St) and Subtitles (/Ss)

■ Options

/St[ ]"*title*"
/Ss[ ]"*subtitle*"

The **/St** and **/Ss** options set the title and subtitle, respectively, for source listings. The quotation marks (" ") around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between **/St** and "*title*", or **/Ss** and "*subtitle*", is optional.

The *title* appears in the upper-left corner of each page of the source listing. The *subtitle* appears below the title.

The **/St** or **/Ss** option applies to the remainder of the command line or until the next occurrence of **/St** or **/Ss** on the command line. These options do not cause source listings to be created. They take effect only when the **/Fs** option is also used to create a source listing.

Both the **/St** and **/Ss** options can be overridden by metacommands in the source file. These options have the same effect as that of **$TITLE** and **$SUBTITLE** metacommands at the top of the file being compiled. If additional **$TITLE** or **$SUBTITLE** metacommands appear in the file being compiled, the title or subtitle is changed accordingly.

The **$TITLE** or **$SUBTITLE** metacommands in a particular file affect only that file and do not change the effects of **/St** or **/Ss** on any other files on the command line.

■ Examples

```
FL /St"INCOME TAX" /Ss4-14 /Fs TAX*.FOR
```

The example above compiles and links all source files beginning with `TAX` and ending with the default extension (**.FOR**) in the current working directory. Each page of the source listing contains the title `INCOME TAX` in the upper-left corner. The subtitle `4-14` appears below the title on each page.

```
FL /c /Fs /St"CALC PROG" /Ss"COUNT" CT.FOR /Ss"SORT" SRT.FOR
```

The example above compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title CALC PROG.

3.3.7.5 Formats for Listings

This section describes and shows examples of the five types of listings available with the **FL** command. See Section 3.3.7.1, "Types of Listings," for information on how to create these listings.

Source Listing

Source listings are helpful in debugging programs as they are being developed. These listings are also useful for documenting the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with expanded include files and any error messages that occurred. If the source file compiles with no errors more serious than warning errors, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each procedure. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of the segment sizes in your program. This summary is useful for analyzing the memory requirements of your program.

Any error messages that occurred during compilation appear in the listing after the line that caused the error, as shown in the following example:

```
      9    hyp = sqrt((sidea**2) + (sideb**2)
***** sqroot.for(9) : error F2115: syntax error
```

The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing.

The example below shows the source listing for a simple FORTRAN program:

```
                                                          PAGE   1
                                                          10-17-86
                                                          18:20:36

  Line#  Source Line          Microsoft FORTRAN Compiler Version 4.00

     1   common a
     2   dimension a(10)
     3   real x
     4   complex c
     5   real *8 d
     6   complex *16 e
     7   character *50 f
     8   integer*2 j
     9   parameter (d=123456789.00056, e=-(.00000122, 1234354 e5))
    10   parameter (f='Note that character strings will be truncated')
    11   parameter (x=1.2345)
    12   parameter (c=(.12345, 123456.789), i = 123, j = 100)
    13   end

main  Local Symbols

Name                 Class    Type              Size    Offset

A . . . . . . . . . . COMMQQ  REAL*4              40      0000

Parameter Symbols        Type                Value

X . . . . . . . . . . . REAL*4             1.2345001E+000
C . . . . . . . . . . . COMPLEX*8        ( 1.2345000E-001, 1.2345679E+005)
D . . . . . . . . . . . REAL*8             1.2345679E+008
E . . . . . . . . . . . COMPLEX*16       (-1.2199999E-006,-1.2343540E+011)
F . . . . . . . . . . . CHARACTER        Note that character strings will
J . . . . . . . . . . . INTEGER*2                    100
I . . . . . . . . . . . INTEGER*4                    123


Global Symbols

Name                       Class    Type              Size    Offset

COMMQQ. . . . . . . . . . common                      40      0000
main. . . . . . . . . . . FSUBRT  ***                 ***     0000

Code size = 0018 (24)
Data size = 0000 (0)
Bss size  = 0000 (0)

No errors detected
```

The Name column lists each global symbol, external symbol, and statically allocated variable declared in the source file. The Parameter Symbols column lists each symbolic constant defined in a **PARAMETER** statement.

For items other than functions and subroutines, the Class column contains either global, common, extern, equiv, or local, depending on how the symbol was defined in the source file. For functions and subroutines, the Class column contains the abbreviations shown in the following list:

Type	Abbreviation
Far function	**FFUNCT**
Near function	**NFUNCT**
Far subroutine	**FSUBRT**
Near subroutine	**NSUBRT**

The Type column shows a simplified version of the symbol's type as declared in the source file. The Type entry for functions is the type declared in the source file.

The Size column is used only for variables. This column specifies the number of bytes of storage allocated for the variable. Note that the amount of storage allocated for an external array may be unknown, so its Size field may be undefined.

The Offset column is used only for symbols with an entry of global or local in the Class field. For variables, the Offset column gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker, in general, combines several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables.

The Value field appears only for parameter symbols. It gives the value of each symbolic constant. Character constants longer than 33 characters are truncated to 33 characters.

The last table in the source listing shows the segments used and their size, as shown below:

```
Code size = 0095 (149)
Data size = 003c (60)
Bss size  = 0000 (0)
```

The byte size of each segment is given first in hexadecimal, and then in decimal (in parentheses). See Section 11.2.2 for a description of the segment model.

Object-Listing File

The object-listing file contains the machine instructions and assembly code for your program. The line numbers are shown in the listing as comments. The machine instructions are on the left and assembly code on the right, as shown in the sample below:

```
SQRT_TEXT       SEGMENT
; Line 6
     PUBLIC     _main
_main       PROC FAR
     *** 000000     55                      push bp
     *** 000001     8b ec                   mov  bp,sp
     *** 000003     b8 02 00                mov  ax,2
     *** 000006     9a 00 00 00 00          call __chkstk
     *** 00000b     9b d9 06 00 00          fld  $T20002
     *** 000010     9b d9 1e 02 00          fstp $S14_SIDEA
; Line 7
     *** 000015     9b d9 06 04 00          fld  $T20003
     *** 00001a     9b d9 1e 06 00          fstp $S15_SIDEB
; Line 9
     *** 00001f     9b d9 06 08 00          fld  $T20004
     *** 000024     9a 00 00 00 00          call __FIsqrt
     *** 000029     9b d9 1e 0a 00          fstp $S16_HYP
     *** 00002e     90 9b                   fwait
```

Assembly-Listing File

The assembly-listing file contains the assembly code corresponding to your program file, as shown below:

```
SQRT_TEXT       SEGMENT
; Line 6
      PUBLIC    _main
_main       PROC FAR
      push bp
      mov  bp,sp
      mov  ax,2
      call __chkstk
      fld  $T20002
      fstp $S14_SIDEA
; Line 7
      fld  $T20003
      fstp $S15_SIDEB
; Line 9
      fld  $T20004
      call __FIsqrt
      fstp $S16_HYP
      fwait
.
.
.
_main       ENDP
SQRT_TEXT ENDS
END
```

Note that the sample shows the same code as in the object listing sample, except that the machine instructions are omitted. This is to ensure that the listing is suitable as input for the Microsoft Macro Assembler (**MASM**).

Combined Source and Object Listing

The combined source and object listing shows one line of your source program followed by the corresponding line (or lines) of machine instructions, as in the following sample:

```
SQRT_TEXT       SEGMENT
;*** c   This program calculates the length of the hypotenuse of a
;*** c   right triangle given the lengths of the other two sides.
;***
;***       real sidea, sideb, hyp
;***
;***       sidea = 3.
; Line 6
      PUBLIC    _main
_main       PROC FAR
      *** 000000      55                    push bp
      *** 000001      8b ec                 mov  bp,sp
      *** 000003      b8 02 00              mov  ax,2
      *** 000006      9a 00 00 00 00        call __chkstk
      *** 00000b      9b d9 06 00 00        fld  $T20002
      *** 000010      9b d9 1e 02 00        fstp $S14_SIDEA
;***       sideb = 4.
; Line 7
      *** 000015      9b d9 06 04 00        fld  $T20003
      *** 00001a      9b d9 1e 06 00        fstp $S15_SIDEB
;***
;***       hyp = sqrt(sidea**2 + sideb**2)
; Line 9
      *** 00001f      9b d9 06 08 00        fld  $T20004
      *** 000024      9a 00 00 00 00        call __FIsqrt
      *** 000029      9b d9 1e 0a 00        fstp $S16_HYP
      *** 00002e      90 9b                 fwait
;***
;***       write(*,100) hyp
.
.
.
_main       ENDP
SQRT_TEXT ENDS
END
;***
```

Note that this sample is like the object-listing sample, except that the program source line is provided in addition to the line number.

Map File

The map file contains a list of segments in order of their appearance within the load module. An example is shown below:

```
Start   Stop    Length Name                    Class
00000H 00059H 0005AH SQRT_TEXT                CODE
0005AH 018E1H 01888H _TEXT                    CODE
.
.
.
```

The information in the `Start` and `Stop` columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The `Length` column gives the length of the segment in bytes. The `Name` column gives the name of the segment, and the `Class` column gives information about the segment type.

The starting address and name of each group appear after the list of segments. A sample group listing is shown below:

```
Origin   Group
0643:0   DGROUP
```

In the example above, **DGROUP** is the name of the data group. **DGROUP** is the only group used by programs compiled with the Microsoft FORTRAN Compiler, Version 4.0.

The map file shown below contains two lists of global symbols: the first list is sorted by symbol address and the second is alphabetical by symbol name. The notation `Abs` appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the FORTRAN compiler. These symbols usually begin with one or two leading underscores or end with **QQ**.

```
Address          Publics by Name

0005:1594        $i8_output
0005:1855        $i8_tpwr10
0000:FE32  Abs   FIARQQ
0000:0E32  Abs   FICRQQ
0000:5C32  Abs   FIDRQQ
0000:1632  Abs   FIERQQ
0000:0632  Abs   FISRQQ
0000:A23D  Abs   FIWRQQ
0000:4000  Abs   FJARQQ
0000:C000  Abs   FJCRQQ
0000:8000  Abs   FJSRQQ
018E:190B        ICLRER
018E:1932        IGETER
0643:058D        OFF_ARGPTR
0643:058B        OFF_DESCPT
0643:00F4        STKHQQ
0005:0885        _access
0005:106F        _brkctl
0005:091A        _chsize
.
.
.
0643:00F0        __aaltstkovr
0643:0278        __abrkp
0643:0228        __abrktb
0643:0278        __abrktbe
.
.
.
```

Finally, the map file gives the program entry point, as shown in the following example:

```
Program entry point at 0005:03C9
```

3.3.8 Searching for Include Files (/I, /X)

■ Options

/I*directory* [[**/I***directory...*]]
/X

The **/I** and **/X** options temporarily override or change the effect of the environment variable **INCLUDE**. These options let you give a particular file or files special handling without changing the compiler environment you normally use. (See Section 2.5, "The Compiler Environment," for a discussion of environment variables.)

The **/I** (for "Include") option adds to the list of standard places for include files (that is, files inserted into a source file using the **$INCLUDE** metacommand). This option causes the compiler to search the directory or directories you specify *before* it searches the standard places given by the **INCLUDE** environment variable.

You can add more than one include-file directory by giving the **/I** option more than once in the **FL** command. The directories are searched in order of their appearance in the command line. Each occurrence of an **/I** option applies only to source files following the option on the command line.

The directories are searched only until the include file specified in the source file is found. If the file cannot be found, the compiler prints an error message and stops processing. When this occurs you must restart compilation with a corrected directory specification.

The following list describes the compiler's search order for include files:

1. The "parent" file's directories. The parent file is defined as the file containing the **$INCLUDE** metacommand. For example, if a file named FILE1 includes a file named FILE2, FILE1 is the parent file of FILE2.

   Include files can be nested; thus, in the preceding example, FILE2 can include another file named FILE3. In this case, FILE1 is said to be the "grandparent" file of FILE3. For nested include files, the search begins with the directories of the parent file, then proceeds through the directories of each of its grandparent files. (See the "Examples" section below for an illustration of this procedure.)

2. The directories specified in each **/I** option.

3. The standard places specified in the **INCLUDE** environment variable.

The **$INCLUDE** metacommand may give a full or partial path specification for the file. (A full path specification starts with the drive name; a partial path specification gives one or more directory names before the name of the file, but does not give a drive name.) If a full path specification is given for the include file, the compiler uses the given path to find the file, and the **INCLUDE** environment variable and any **/I** options have no effect. If a partial path specification is given, the compiler attempts to find that path, starting from the parent file's directory, then from the grandparent files' directories, then from the directories given on the command line, and finally from the directories given by the **INCLUDE** environment variable.

The **/X** (for "Exclude") option prevents the compiler from searching the standard places given by the **INCLUDE** variable. When **/X** is given, **FL** considers the list of standard places to be empty. The parent and grandparent directories will still be searched, however.

Like the **/I** option, **/X** applies only to source files following the option on the command line. The **/X** option can be followed by one or more **/I** options; this causes the compiler to search only the parent and grandparent directories and the directories given by the **/I** options, ignoring the standard places.

■ Examples

```
FL /IC:\TESTDIR /IC:\PREVIOUS *.FOR
```

The example above assumes that the **INCLUDE** environment variable is set to `C:\FOR\INCLUDE`. It compiles all source files with the default extension (**.FOR**) in the current working directory, searching for include files in the following order:

1. The current working directory
2. `\TESTDIR`, the first directory on the command line
3. `\PREVIOUS`, the second directory on the command line
4. `\FOR\INCLUDE`, the directory given by the **INCLUDE** environment variable

However, if the metacommand $INCLUDE:'\SUB\DEFS' is contained in one of the source files, the compiler adds the subdirectory \SUB to the end of each path it searches. Thus, the search for the include file named DEFS proceeds in the following order:

1. The current working directory (which contains any parent source files)
2. The \SUB subdirectory of the current working directory
3. \TESTDIR (the first directory on the command line)
4. \PREVIOUS (the second directory on the command line)
5. \FOR\INCLUDE (the directory given by the **INCLUDE** environment variable)

```
FL ..\TESTS\*.FOR
```

The example above assumes that the **INCLUDE** environment variable is set to C:\FOR\INCLUDE. It compiles all source files with the default extension (**.FOR**) in the directory named ..\TESTS, searching directories for include files in the following order:

1. ..\TESTS (the directory containing any possible parent files)
2. \FOR\INCLUDE (the directory given by the **INCLUDE** environment variable)

However, if one of the source files in the directory ..\TESTS contains the metacommand $INCLUDE:'\SUB\DEFS', the compiler adds the subdirectory \SUB to the end of each path it searches. Thus, the search for the include file named DEFS proceeds in the following order:

1. ..\TESTS\SUB (adding the subdirectory \SUB to the directory ..\TESTS, where ..\TESTS is the directory containing the parent source file)
2. \FOR\INCLUDE\SUB (adding the subdirectory \SUB to the directory \FOR\INCLUDE, where \FOR\INCLUDE is the directory given by the **INCLUDE** environment variable)

If the file \SUB\DEFS contains the metacommand $INCLUDE:'COMS', the compiler searches directories for the nested include file named COMS in the following order:

1. ..\TESTS\SUB (the directory containing DEFS, the parent file of the file named COM)
2. ..\TESTS (the directory containing the grandparent source file of the file named COM)
3. ..\FOR\INCLUDE (the directory given by the **INCLUDE** environment variable)

In this last case, since COMS is not specified as part of another subdirectory, no subdirectory is added to the end of the path specified in the **INCLUDE** environment variable.

The search ends as soon as the file is found.

```
FL MAIN.FOR /X /ITEST1 SUB1.FOR /ITEST2 SUB2.FOR
```

The example above uses a combination of the **/I** and **/X** options to control the search path. Since no **/I** option appears before MAIN.FOR on the command line, the compiler searches for any files included by MAIN.FOR in the standard places defined by the **INCLUDE** variable (after searching the parent file's directory). Since the **/X** option precedes the next file name, SUB1.FOR, the compiler does not search the standard places for any files SUB1.FOR includes (in this case, the environment variable is not used). Instead, only the directory of the parent source file SUB2.FOR and the directory TEST1 are searched. If the include file or files cannot be found in one of those places, an error occurs. The second **/I** option adds one more directory to be searched for any include files specified in the parent file SUB2.FOR. The TEST2 subdirectory is searched *after* the TEST1 subdirectory.

3.3.9 Handling Warnings and Errors

You may encounter several different kinds of error messages when you compile, link, and run a Microsoft FORTRAN program. Section 3.3.9.1 gives an overview of Microsoft FORTRAN error messages. Several options are available to control the types of warnings generated at compile time and to enable or disable expanded error handling at run time. See Sections 3.3.9.2–3.3.9.4 for a description of these options.

3.3.9.1 Understanding Error Messages

Error messages can appear at several different stages of program development:

- In the compiling stage, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
- During the linking stage, the linker is responsible for generating error messages.
- During program execution, any error messages you see are run-time error messages. This category includes messages about floating-point exceptions, which are errors generated by an 8087 or 80287 coprocessor.

Other utilities included in this package, such as **MAKE** and **EXEMOD**, generate their own error messages. See Appendix E, "Error Messages and Limits," for a complete list of error messages.

When you are compiling and linking using the **FL** command, you may see both compiler and linker messages. The **LINK** program banner appears on the screen when the linking process begins. Compiler messages are any messages that appear before the **LINK** banner, and linker messages are those that appear after the banner. Compiler messages have numbers preceded by the letter F, and linker messages have numbers preceded by the letter L.

You can also distinguish the type of a message by its format. See Appendix E for a description of error-message formats, a list of actual error messages, and explanations of the circumstances that cause them.

Compiler error messages are sent to the standard output, which is usually your terminal. You can redirect the messages to a file or printer by using one of the DOS redirection symbols: > or >>.

Note that not all error messages are sent to the standard output; run-time error messages, for example, are sent to the standard error output. You can use the **ERROUT** utility (described in Section 7.5) to specify redirection of errors that are sent to the standard error output.

Error redirection is especially useful in batch-file processing. For example, the following command redirects error messages to the printer device (designated by **PRN**):

```
FL /c COUNT.FOR > PRN
```

See Section 3.3.7.2, "Special File Names," or your DOS documentation for a list of device names, including **PRN**.

In the following command, only output that ordinarily goes to the console screen is redirected.

```
FL COUNT.FOR > COUNT.ERR
```

The **FL** control program returns an exit code that indicates the status of the compilation. Exit codes are useful with the DOS batch command **IF ERRORLEVEL** and with the **MAKE** utility. They allow you to test for the success or failure of the compilation before proceeding with other tasks. See Appendix B, "Using Exit Codes," for more information.

3.3.9.2 The Debug (/4Yb, /4Nb) and Declare (/4Yd, /4Nd) Options

The debug and declare options control extended error handling and warnings about undeclared variables. The **/4** prefix identifies these options to the **FL** command as FORTRAN-specific options.

You can specify more than one option following the **/4**; for example, the **/4Ybd** option would enable extended error handling and warnings for undeclared variables. You can also include the source-file syntax options, discussed in Section 3.3.11, in the same **/4** option.

The Debug Option

- **Option**

/4{Y | N}b

The debug option controls extended error handling at run time. When enabled, the debug option provides information to be used by the error-handling system in the program file. See the discussion of the **$DEBUG** metacommand in Section 6.2.1 of the *Microsoft FORTRAN Compiler*

Language Reference for a description of the types of errors that are detected in extended error handling. When the debug option is enabled, loop optimization in the program is disabled.

Debugging is enabled by giving the **/4Yb** option (**Y** for "Yes") and disabled with **/4Nb** (**N** for "No"). By default, debugging is disabled.

The **/4Yb** or **/4Nb** option applies to the remainder of the command line or until the next occurrence of **/4Yb** or **/4Nb** on the command line. These options have no effect on object files specified on the command line.

The debug option can be combined with other options that begin with **/4** and either **Y** or **N**. For example, `/4Ybd` turns on both the debug and the declare options.

The debug option has the same effect as a **$DEBUG** or **$NODEBUG** metacommand appearing at the top of the source file being compiled. If a **$DEBUG** or **$NODEBUG** metacommand appears later in the file being compiled, debugging for that file is enabled or disabled, as appropriate.

The **$DEBUG** and **$NODEBUG** metacommands in a particular file affect only that file and do not change the effects of **/4Yb** or **/4Nb** on any other files on the command line.

The **/4Yb** option does not accept a string argument for conditional compilation. Use the **/4cc** option, described in Section 3.3.12, instead.

■ Examples

```
FL MAIN.FOR /4Yb /Fs TEST.FOR
```

The example above compiles and links two files. Debugging is enabled for `TEST.FOR`, and a source listing named `TEST.LST` is created. Neither the debugging option nor the source-listing option applies to `MAIN.FOR`.

```
FL /c /4Ybd ONE.FOR /4Nd TWO.FOR
```

The example above compiles `ONE.FOR` with both the debug and declare options enabled. (The following section describes the declare option.) The declare option is disabled when compiling `TWO.FOR`, but the debug option is still in effect.

The Declare Option

■ Option

/4{Y | N}d

The declare option controls warnings about undeclared variables. When the declare option is enabled, the compiler generates a warning message at the first use of any variable which has not been declared in a type statement.

The **/4Yd** option (**Y** for "Yes") enables the declare option; **/4Nd** (**N** for "No") disables it. The declare option is disabled by default (unless a **$DECLARE** metacommand occurs in the source file).

The **/4Yd** or **/4Nd** option applies to the remainder of the command line or until the next occurrence of **/4Yd** or **/4Nd** on the command line. These options have no effect on object files given on the command line.

The declare option can be combined with other options that begin with **/4** and either **Y** or **N**. For example, `/4Ybd` turns on both the debug and the declare options.

The declare compiler option provides the same effect as a **$DECLARE** or **$NODECLARE** metacommand appearing at the top of each source file being compiled. If **$DECLARE** or **$NODECLARE** metacommands appear later in the file being compiled, warnings are enabled or disabled, as appropriate. Note that if the source file being compiled contains a **$DECLARE** or **$NODECLARE** metacommand at the top of the file, the **/4Yd** or **/4Nd** option has no effect.

The **$DECLARE** and **$NODECLARE** metacommands in a particular file affect only that file and do not change the effects of **/4Yd** or **/4Nd** on any other files on the command line.

■ Examples

```
FL /4Ybd *.FOR > DECLARE
```

The example above compiles and links all source files with the default extension (**.FOR**) in the current working directory. The debug and declare options are both enabled. All messages (including warnings about undeclared variables) are redirected to the file DECLARE.

```
FL /4Yb ONE.FOR /4Yd TWO.FOR
```

The example above turns on debugging for both ONE.FOR and TWO.FOR; the declare option is also enabled for TWO.FOR.

3.3.9.3 The Warning-Level Option (/W)

■ Option

/W{0 | 1}

You can suppress warning messages produced by the compiler by using the **/W** (for "Warning") option. Compiler warning messages are any messages beginning with F4; see Appendix E for a full listing of these messages. Warnings indicate potential problems, rather than actual errors, with statements that may not be compiled as you intend.

/W1 (the default) causes the compiler to display warning messages. **/W0** turns off warning messages. The **/W0** option is useful when compiling programs that deliberately include questionable statements.

/W0 applies to the remainder of the command line or until the next occurrence of **/W1** on the command line. These options have no effect on object files given on the command line.

■ Example

```
FL /W0 CRUNCH.FOR PRINT.FOR
```

This example suppresses warning messages when the files CRUNCH.FOR and PRINT.FOR are compiled.

3.3.9.4 Syntax Errors (/Zs)

- **Option**

/Zs

The **/Zs** option causes the compiler to perform a syntax check only. This option provides a quick way to find and correct syntax errors before you try to compile a source file. With **/Zs**, no code is generated and no object files or object listings are produced. However, you can specify the **/Fs** option on the same command line to generate a source listing.

The **/Zs** option applies to all source files that follow the option on the command line but does not affect any source files preceding the option.

- **Example**

```
FL /Zs TEST*.FOR
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with TEST and end with the default extension (**.FOR**). The compiler displays messages for any errors found.

3.3.10 Setting the Default Integer Size (/4I2, /4I4)

- **Option**

/4I{2 | 4}

The **/4I** option allocates either 2 or 4 bytes of memory for all variables declared in the source file as **INTEGER** or **LOGICAL** variables. The default allocation is 4 bytes.

The **/4I** option applies to the remainder of the command line or until the next occurrence of **/4I** on the command line.

The **/4** prefix identifies this option to the **FL** command as a FORTRAN-specific option.

The **/4I** option has the same effect as a **$STORAGE** metacommand at the top of each file being compiled. If a **$STORAGE** metacommand already appears in the file being compiled, the size given by the metacommand is used. The **$STORAGE** metacommand in a particular file affects only that file and does not change the effects of **/4I** on any other files on the command line. See the *Microsoft FORTRAN Compiler Language Reference* for more information on the **$STORAGE** metacommand.

Note

If you use this option, you must declare explicit lengths for any integer or logical variables that are associated in **EQUIVALENCE** statements, since this option causes such variables to be placed in different memory locations.

■ **Example**

```
FL /4I2 /FeTESTPROG *.FOR
```

This example allocates 2 bytes of memory for **INTEGER** and **LOGICAL** variables when compiling and linking all source files in the current working directory. The executable file is named `TESTPROG.EXE`.

3.3.11 Controlling Source-File Syntax (/4Y6, /4N6, /4Yf, /4Nf, /4Ys, /4Ns, /4Yt, /4Nt)

/4{Y | N}6 Enables (**Y**) or disables (**N**) FORTRAN 66-style **DO** statements.
/4{Y | N}f Enables (**Y**) or disables (**N**) free-form format.
/4{Y | N}s Enforces strict syntax (**Y**) or allows extensions (**N**).
/4{Y | N}t Enables (**Y**) or disables (**N**) truncation of variable names.

The options described in this section give you control over the statements and structure permitted in a source file. Each option is enabled by using **Y** (for "Yes") or disabled by using **N** (for "No"). In these options, the **/4** prefix identifies them to the **FL** command as FORTRAN-specific options. You can specify more than one option following the **/4**; for example, the **/4Yft** option would enable free-form programs and truncation of variable names. You can also include the debug and declare options, discussed in Section 3.3.9.2, in the same **/4** option.

These options correspond to FORTRAN metacommands, which are described in detail in the *Microsoft FORTRAN Compiler Language Reference*. The following list gives the metacommand corresponding to each option and identifies the default option:

Option	Metacommand
/4Y6	**$DO66**
/4N6 (default)	None
/4Yf	**$FREEFORM**
/4Nf (default)	**$NOFREEFORM**
/4Ys	**$STRICT**
/4Ns (default)	**$NOTSTRICT**
/4Yt (default)	**$TRUNCATE**
/4Nt	**$NOTRUNCATE**

Any of these options can be combined with other options that begin with **/4** and either **Y** or **N**. For example, `/4Yf6` enables both free-form format and FORTRAN 66-style **DO** statements.

When one of these **/4** options appears on the **FL** command line, it applies to the remainder of the command line or until another occurrence of a **/4** option on the command line reverses its effect.

Each option has the same effect as placing the corresponding metacommand at the top of the source file being compiled. If a conflicting metacommand occurs later in the source file, the metacommand in the source file overrides the effect of the command-line option for that file. Any option can be used with a file that already contains the corresponding metacommand without causing an error.

The metacommands in a particular file affect only that file and do not change the effects of these **/4** options on any other files on the command line.

■ Examples

```
FL /c /4Yds TEST.FOR /4Nd STABLE.FOR
```

The command line above causes **FL** to compile TEST.FOR using strict FORTRAN 77 syntax (disallowing all Microsoft extensions). The declare option is also enabled, so use of undeclared variables produces warning messages. When the second file, STABLE.FOR, is compiled, the strict option is still in effect, but the declare option is disabled.

```
FL /4Yf /4Nt *.FOR
```

The command line above enables free-form format and disables truncation of variable names when compiling and linking all source files in the current working directory.

3.3.12 Conditional Compilation (/4cc)

■ Option

/4cc*string*

The **/4cc** option permits conditional compilation of a source file. The *string* is a set of alphabetic characters controlling which lines in the source file are to be compiled. The **/4cc** option applies to any source files following the option on the command line.

Any source file line that begins with a letter found in *string* is compiled; lines beginning with other letters are treated as comments. Case is not significant. The letter must appear in column 1 of the source-file line.

The *string* can be enclosed in double quotation marks (" ") if desired, but the quotation marks are not required.

Note

Program lines with the character **C** or **c** in column 1 are always treated as comments.

■ Example

```
FL /c /4ccXYZ PRELIM.FOR
```

This example includes all lines beginning with X, Y, or Z in the compilation of the source file PRELIM.FOR.

3.3.13 Preparing for Debugging (/Zi, /Od, /Zd)

■ Options

/Zi Prepares for debugging with the Microsoft CodeView debugger
/Od Disables optimization
/Zd Prepares for debugging with **SYMDEB**

The **/Zi** option produces an object file containing full symbolic debugging information—including the symbol table and line numbers—for use with the Microsoft CodeView window-oriented debugger.

When you use the **FL** command to compile and link, giving the **/Zi** option automatically causes the **/CO** option to be given at link time. If you link in a separate step (using either **FL** or the **LINK** command) instead of compiling and linking in one step, be sure to give the **/CO** option when you link. Otherwise, symbols and source-code lines will be missing when you run the CodeView debugger. See Section 4.6.17, "Preparing for Debugging," for more information on **/CO**.

The **/Od** option tells the compiler not to perform optimization. Without the **/Od** option, the default is to optimize. Using **/Od** is recommended whenever you use **/Zi**. It is also recommended while testing, since it can improve compilation speed by 30 to 35 percent. See Section 3.3.15 for information about the **/Od** option.

Note

If you use **/Od** to compile, the **F3S.EXE** file must be in the current search path.

Since optimization can involve rearrangement of instructions and storage of values in machine registers, it may be difficult to recognize and correct your code if you optimize before debugging.

Note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Note

When the debug option (**/4Yb**) is enabled, loop optimization is disabled. See Section 3.3.9.2 for a description of the debug option.

See Section 3.3.15, "Optimizing," for a discussion of additional optimization options.

The **/Zd** option produces an object file containing line-number records that correspond to the line numbers of the source file. The **/Zd** option is useful when you want to pass an object file to the **SYMDEB** symbolic debugger, available with other Microsoft products. The debugger can use the line numbers to refer to program locations. However, only global symbol-table information is available with **SYMDEB** (unlike the CodeView debugger, which also recognizes local symbols).

When you use the **FL** command to compile and link, giving the **/Zd** option automatically causes the **/LI** option to be given at link time. (See Section 4.6.6, "Including Line Numbers in the Map File," for more information on **/LI**.) If you compile a source file with the **/Zd** option, then link in a separate step using **FL**, be sure to give the **/Zd** option when you link. (If you link using the **LINK** command, give the **/LI** option.) Otherwise, your executable file will not contain line numbers.

The **/Zd** option automatically generates a map file, whether or not the **/Fm** option is given. If **/Fm** is not used to specify a file name or location for the file, the map file is created in the current working directory and given the default name, as described in Section 3.3.7.1, "Types of Listings."

The **/Zi**, **/Od**, and **/Zd** options apply to any source files following the option on the command line, but do not affect source files preceding the option. The **/Zi** and **/Od** options have no effect on object files given on the command line. **/Zd** causes the **/LI** option to be given at link time.

■ Example

```
FL /Zi /Od /Fs P*.FOR /FePROCESS /FmPROCESS
```

This command compiles all source files in the current working directory beginning with P and ending with the default extension (**.FOR**), creating object files that contain the symbolic information required for debugging with the CodeView debugger. Optimization is disabled with **/Od**. The **/Fs** option creates a source listing for each source file. The executable file is named PROCESS.EXE, and a map file named PROCESS.MAP is also created.

3.3.14 Using an 80186, 80188, or 80286 Processor (/G0, /G1, /G2)

■ Options

/G0	8086/8088 instruction set (default)
/G1	80186/80188 instruction set
/G2	80286 instruction set

If you have an 80186, 80188, or 80286 processor, you can use the **/G1** or **/G2** option to enable the instruction set for your processor. Use **/G1** for the 80186 or 80188 processor; use **/G2** for the 80286. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, do not use the **/G1** or **/G2** option.

The **/G0** option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly since the 8086/8088 instruction set is used by default. Programs compiled this way also run on an 80186, 80188, or 80286 processor.

Only one of these three options is allowed on the **FL** command line. If more than one appears, **FL** issues a warning and generates code using the **/G1** option.

■ **Example**

```
FL /G2 /FeFINAL *.FOR
```

The example above compiles and links all source files with the default extension (**.FOR**) in the current working directory, using the 80286 instruction set. The resulting program, named FINAL.EXE, will run only on an 80286.

3.3.15 Optimizing (/O Options)

The optimizing procedures available with the Microsoft FORTRAN Compiler can reduce the storage space and execution time required for a compiled program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the **/O** options to exercise greater control over the optimization performed.

■ **Option**

/O*letters*

The **/O** (for "Optimize") option controls optimization. The *letters* after the **/O** option let you influence how the compiler optimizes your code. The *letters* are one or more of the following:

Character	Optimizing Procedure
x	Full optimization; favors execution time (default)
t	Equivalent to **x**
s	Favors code size
d	Disables optimization; leaves stack checking on
p	Improves consistency of floating-point results

The letters can appear in any order; for example, **/Osp** and **/Ops** have the same effect. Only one **/O***letters* option can appear on the **FL** command line and, regardless of its position, it applies to all source files on the line.

When you do not give an **/O** option to the **FL** command, or when you give an **/O** option but do not use the letter **x**, the compiler automatically uses **/Ox**, meaning that the program is optimized for maximum execution speed. The **/Ot** option is equivalent to the **/Ox** option. Whenever the compiler has a choice between producing smaller (but perhaps slower) code and larger (but perhaps faster) code, the compiler chooses to generate the larger, faster code. To cause the compiler to produce smaller code instead, use the **/Os** option.

The **/Od** option turns off optimization. This option is useful in the early stages of program development because it avoids optimizing code that will be changed and improves compilation speed by approximately 30 to 35 percent. Because optimization may involve rearrangement of instructions, you may also want to specify the **/Od** option when you use a debugger other than the CodeView debugger with your program, or when you want to examine an object listing. (The **/Zi** option, which prepares a program for debugging with the CodeView debugger, automatically turns off loop optimization and optimization involving code rearrangement.) If you optimize before debugging, it can be difficult to recognize and correct your code.

Note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Note

When the debug option (**/4Yb**) is enabled, loop optimization is disabled. See Section 3.3.9.2 for a description of the debug option.

The **/Op** is useful when floating-point results must be consistent within a program. This option changes the way in which the compiler handles floating-point values by default. Ordinarily, the compiler waits to assign floating-point values whenever this is possible. Instead, the compiler stores each floating-point value in an 80-bit machine register. For further calculations involving that value, the compiler uses the value in the register; it may not assign the final result to a variable until some or all operations using that value are complete. However, since most floating-point types are allocated less than 80 bits of storage (32 bits for the **REAL*4** type and 64 bits for the **REAL*8** type), the value stored in the register may actually be more precise than the same value stored in a floating-point variable. Over the course of a program, the value that results from this use of the machine register may be quite different from the value that would result if the compiler assigned the result of each operation to a variable and used the variable in later calculations. If you specify the **/Op** option, the compiler uses the variable method instead of the default machine-register method for handling floating-point values.

Using **/Op** gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more control over the truncation (and hence the consistency) of floating-point values.

■ Examples

```
FL /c /Os FILE.FOR
```

The command above favors code size over execution speed when compiling FILE.FOR.

```
FL /Od *.FOR
```

The command above compiles and links all FORTRAN source files with the default extension (**.FOR**) in the current directory and disables optimization. This command might be useful during the early stages of program development, since it improves compilation speed.

```
FL /Op /FeTESTRUN *.FOR
```

The command above causes floating-point assignments to variables to be carried out immediately (where specified) when compiling all source files with the default extension (**.FOR**) in the current working directory. By default, execution time is favored in the optimization. The resulting program is named TESTRUN.EXE.

3.3.16 Enabling Stack Probes (/Ge)

■ Option

/Ge

You can check your program for stack-overflow errors by enabling stack probes with the **/Ge** option. A stack probe is a short subroutine called on entry to a subroutine to verify that the program stack has enough space for local variables to be allocated. When stack probes are enabled, the stack-probe subroutine is automatically called at every entry point. The stack-probe subroutine generates a message and ends the program if it determines that the required stack space is not available. (By default, no calls to the stack-probe subroutine are made.)

The **/Ge** option is useful if you are not sure whether or not your program exceeds the available stack space. For example, stack probes may be needed for programs that make a large number of subroutine calls.

The **/Ge** option applies to any source files following the option on the command line.

Note

Although the default option, leaving stack probes disabled, reduces program size, it means that no compiler error message is displayed if a stack overflow occurs. You may want to use the **/Ge** option during program testing to make sure that the program does not cause a stack overflow.

■ Example

```
FL /c /Ge /Ot FILE.FOR
```

This example enables stack probes and favors execution time when compiling `FILE.FOR`.

3.3.17 Suppressing Automatic Library Selection (/Zl)

■ Option

/Zl

Ordinarily the compiler places in the object file the name of the FORTRAN library corresponding to the floating-point and memory-model options you choose. The linker uses the library name to link the program automatically with the corresponding library. Thus, you do not need to specify a library name to the linker, provided that the appropriate library exists for the floating-point and memory-model options you are using.

The **/Zl** option suppresses this use of library names in object files. When you specify **/Zl**, the compiler does not place a library name in the object file. As a result, the object file is slightly smaller.

The **/Zl** option is useful when you are building your own library of routines, since not every routine in the library is required to contain the library information. Although the **/Zl** option saves only a small amount of space for a single object file, the total space saving is significant in a library containing many object modules. When you link a library of object modules created *with* the **/Zl** option with a program file compiled *without* the **/Zl** option, the program file supplies the library information.

The **/Zl** option applies to the remainder of the source files on the command line.

■ Examples

```
FL ONE.FOR /Zl TWO.FOR
```

The example above creates an object file named ONE.OBJ. Since no floating-point or memory-model options are specified on the **FL** command line, this object file will contain the name of the FORTRAN library that corresponds to the default floating-point and memory-model options (**LLIBFOR7.LIB**). The example also will create an object file named TWO.OBJ that contains no library information, since the **/Zl** option appears before the file name on the command line. When ONE.OBJ and TWO.OBJ are linked to create an executable file, the library information in ONE.OBJ causes **LLIBFOR7.LIB** to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

```
FL /c /Zl *.FOR
```

The example above compiles all source files with the default extension (**.FOR**) in the current working directory. None of the resulting object files contains library information.

3.3.18 Setting the Stack Size (/F)

/F *hexnum*

The **/F** option sets the size of the program stack. A space must separate the **/F** and *hexnum*.

The *hexnum* is a hexadecimal number representing the stack size in bytes. The number must be positive and cannot exceed 10,000 hexadecimal (65,536 decimal). The default stack size is 2K.

This option affects object files only; it does not have any effect on source files.

Using the **/F** option with the **FL** command has the same effect as using the **/STACK** option with the **LINK** program. See Section 4.6.9 for more information on **/STACK** option.

■ Example

```
FL /F C00 *.OBJ
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all object files in the current working directory.

3.3.19 Restricting the Length of External Names (/H)

■ **Option**

/H*number*

The **FL** command allows you to restrict the length of global names by using the **/H** option. The *number* is an integer specifying the maximum number of significant characters in global names.

When you use the **/H** option, the compiler considers only the first *number* characters of global names used in the program. The program may contain global names longer than *number* characters; the extra characters are simply ignored. Any truncation is performed after compilation has completed.

This option has no effect on local names.

3.3.20 Labeling the Object File (/V)

■ **Option**

/V"*string***"**

Use the **/V** (for "Version") option to embed a given text *string* (typically as a label) into an object file. The quotation marks surrounding the string may be omitted if the string does not contain white-space characters.

Object files are machine readable but are not easily read and understood by humans. A common use of the **/V** option is to label an object file with a version number or copyright notice.

The **/V** option applies to any source files following the option on the command line.

■ **Example**

```
FL /V"Microsoft FORTRAN Compiler Version 4.0" MAIN.FOR
```

This command places the string `Microsoft FORTRAN Compiler Version 4.0` in the object file `MAIN.OBJ`.

3.3.21 Compatibility with Version 3.2 (/Gr)

- **Option**

/Gr

The **/Gr** option allows you to create object files that are compatible with Version 3.2 of Microsoft FORTRAN. In object files created by compiling with **/Gr**, the **SI** and **DI** registers do not have to be preserved during subprogram execution. (The default code for Version 4.0 object files preserves the contents of these registers during subprogram execution.)

3.4 Using FL to Link without Compiling

Just as you can use the **FL** command to compile source files without linking the resulting object files (see Section 3.3.4), you can use **FL** just to link object files. If all of the files you give **FL** have extensions other than **.FOR** and if no **/Tf** options appear, **FL** skips the compiling stage and links your files. To link object files, use the following special form of the **FL** command:

FL *objfile*[[,*objfile*...]] **/link** [[*libfield*]] [[*linkoptions*]]

Note

You cannot create an overlaid version of your program with the **FL** command; you must explicitly use the **LINK** command. See Section 4.7 for a description of overlays.

Anytime you use **FL** to link object files, it gives the resulting executable file the base name of the first object file on the command line, plus an **.EXE** extension, by default. (This is the same naming convention that **FL** uses when it compiles source files first, then links the resulting object files.)

The **FL** command options that begin with **/F** allow you to give the file names and options that you would otherwise give on the **LINK** command line (or in response to **LINK** prompts). The following list shows each **FL** option for the linker and the corresponding **LINK** command-line field, prompt, or option:

FL Option	LINK Field / Prompt / Option
/Fe*exefile*	The *exefile* field or "Run File" prompt
/Fm*mapfile*	The *mapfile* field or "List File" prompt
/link *libfield linkoptions*	The *libfiles* field or "Libraries" prompt, and any of the **LINK** options described in Sections 4.6.1 – 4.6.17
F *hexnum*	The **/STACK** option

See Section 3.3.6 for a description of the **/Fe** option, Section 3.3.7.1 for a description of the **/Fm** option, and Section 3.3.18 for a description of the **/F** option. Sections 4.4.1.2 – 4.4.1.5 describe the **LINK** command-line fields, and Section 4.6.9 describes the **/STACK** option.

If you use the **/link** *libfield linkoptions* option with the **FL** command, it must be the last option on the command line. Use this field to specify any of the linker options described in Sections 4.6.1 – 4.6.17.

Chapter 4
Linking

4.1 Introduction 109
4.2 Running the Linker 109
4.3 Understanding LINK Memory Requirements 110
4.4 Linking with the LINK Command 111
4.4.1 Using a Command Line 111
4.4.1.1 Command-Line Defaults 111
4.4.1.2 Specifying Object Files 112
4.4.1.3 Specifying the Executable-File Name 112
4.4.1.4 Specifying a Map File 113
4.4.1.5 Specifying Libraries 113
4.4.1.6 Specifying Linker Options 114
4.4.2 Using Prompts 115
4.4.3 Using a Response File 116
4.5 Linking FORTRAN Program Files 118
4.5.1 The Program Entry Point 118
4.5.2 Specifying File Names 118
4.5.3 Specifying Libraries to Be Searched 119
4.5.3.1 Searching Additional Libraries 119
4.5.3.2 Searching Different Locations for Libraries 120
4.5.3.3 Overriding Libraries Named in Object Files 120
4.6 Using Linker Options 121
4.6.1 Viewing the Options List (/HE) 123
4.6.2 Pausing during Linking (/P) 123

4.6.3 Displaying Linker Process Information (/I) 124
4.6.4 Packing Executable Files (/E) 125
4.6.5 Listing Public Symbols (/M) 126
4.6.6 Including Line Numbers in the Map File (/LI) 126
4.6.7 Preserving Case Sensitivity (/NOI) 127
4.6.8 Ignoring Default Libraries (/NOD) 127
4.6.9 Controlling Stack Size (/ST) 128
4.6.10 Setting the Maximum Allocation Space (/CP) 129
4.6.11 Controlling Segments (/SE) 130
4.6.12 Setting the Overlay Interrupt (/O) 131
4.6.13 Ordering Segments (/DO) 131
4.6.14 Controlling Data Loading (/DS) 132
4.6.15 Controlling Executable-File Loading (/HI) 133
4.6.16 Preserving Compatibility (/NOG) 133
4.6.17 Preparing for Debugging (/CO) 134
4.7 Using Overlays 135
4.7.1 Restrictions on Overlays 136
4.7.2 Overlay-Manager Prompts 136
4.8 Terminating the LINK Session 137
4.9 How the Linker Works 137
4.9.1 Alignment of Segments 138
4.9.2 Frame Number 139
4.9.3 Order of Segments 139
4.9.4 Combined Segments 140
4.9.5 Groups 141
4.9.6 Fixups 141

4.1 Introduction

The Microsoft Overlay Linker (**LINK**) is used to combine object files compiled with the Microsoft FORTRAN Compiler into a single executable file. It can be used with object files compiled or assembled on 8086/8088 or 80286 machines. The format of input to the linker is the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel® 8086 OMF.

The output file from **LINK** (that is, the executable file) is not bound to specific memory addresses. Thus, the operating system can load and execute this file at any convenient address. **LINK** can produce executable files containing up to 1 megabyte of code and data.

The following sections explain how to run the linker and specify options that control its operation.

4.2 Running the Linker

You can run the linker in one of two ways:

1. Use the **FL** command to invoke the linker automatically after the compiling stage, as described in Section 3.2.1. The **/link** option of the **FL** command is used to pass information to the linker, as discussed in Section 3.4, "Using FL to Link without Compiling."

2. Use the **/c** option of the **FL** command to stop processing after compilation; then use the **FL** or **LINK** command in a separate step to link your object files.

   Section 3.4 describes how to use the **FL** command to invoke the linker; Sections 4.2 through 4.4 describe the use of the **LINK** command. The rules for specifying file names and options in both commands are described in Sections 4.5 and 4.6.

4.3 Understanding LINK Memory Requirements

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, **LINK** creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version:

- If the linker is running on DOS Version 3.0 or later, it uses a DOS system call to create a temporary file with a unique name in the current working directory.
- If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named **VM.TMP**.

When the linker creates a temporary disk file, you will see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive letter
```

Here, *tempfile* is ".\" followed by either **VM.TMP** or a name generated by DOS, and *letter* is the current drive. After this message appears, do not remove the disk from the drive specified by *letter* until the link session ends. If the disk is removed, the operation of **LINK** is unpredictable, and you may see the following message:

```
unexpected end-of-file on scratch file
```

When this happens, rerun the link session. The temporary file created by **LINK** is a working file only. **LINK** deletes it at the end of the link session.

Note

Do not give any of your own files the name **VM.TMP**. The linker displays an error message if it encounters an existing file with this file name.

4.4 Linking with the LINK Command

Instead of using the **FL** command to invoke the linker, you can use the **LINK** command to invoke **LINK** directly. You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. This type of file is known as a "response file."

4.4.1 Using a Command Line

Use the following form of the **LINK** command to specify input on the command line:

LINK *objfiles*[[,[[*exefile*]][,[[*mapfile*]][,[[*libfiles*]]]]]] [[*options*]][[;]]

4.4.1.1 Command-Line Defaults

You can select the default for any command-line field by omitting the file name or names before the commas. The only exception to this is the default for *mapfile*: if you use a comma as a placeholder for the map file on the command line, **LINK** creates a map file. This file has the same base name as the executable file. Use NUL for the map-file name if you do not want to produce a map file.

You can also select default responses by using semicolons (;). The semicolon tells **LINK** to use the defaults for all remaining fields.

If you do not give all file names on the command line, or if you do not end the command line with a semicolon, the linker prompts you for the files you omitted, using the prompts described in Section 4.4.2, "Using Prompts."

If you do not specify a drive or directory for a file, the linker assumes that the file is on the current drive and directory. If you want the linker to create files in a different location than the current drive and directory, you must specify the new drive and directory for each such file on the command line.

See Sections 4.4.1.2 through 4.4.1.6 for a description of the input you give in each command-line field. See Section 4.5.2 for a description of the rules for entering file names in the **LINK** command fields.

Since you can specify some of the same information on the **FL** command line, see Section 3.4 for the **FL** options that correspond to **LINK** command fields.

4.4.1.2 Specifying Object Files

- **Field**

objfiles

Use the *objfiles* field to give the names of the object files you are linking. At least one object-file name is required. A space or plus sign (+) must separate each pair of object-file names.

LINK automatically supplies the **.OBJ** extension when you give a file name without an extension. If your object file has a different extension, or if it appears in another directory or on another disk, you must give the full name—including the extension and path name—for the file to be found. If **LINK** cannot find a given object file, it displays a message and waits for you to change disks.

4.4.1.3 Specifying the Executable-File Name

- **Field**

exefile

The *exefile* field allows you to specify the name of the executable file.

If the file name you give does not have an extension, **LINK** automatically adds **.EXE** as the extension. You can give any file name you like; however, if you are specifying an extension, you should always use **.EXE**, because DOS expects executable files to have either this extension or the **.COM** extension.

4.4.1.4 Specifying a Map File

- **Field**

mapfile

The *mapfile* field allows you to specify the name of the map file, if you are creating one.

Also specify the **/MAP** option of the **LINK** command if you want to include public symbols and their addresses in the map file. See Section 4.6.5, "Listing Public Symbols," for a description of the **/MAP** option and Section 3.3.7.5, "Formats for Listings," for a description of map-file formats.

If you specify a map-file name without an extension, **LINK** automatically adds an extension of **.MAP**. **LINK** creates the map file in the current working directory unless you specify a path name for the map file.

4.4.1.5 Specifying Libraries

- **Field**

libfiles

When you compile a source file, the **FL** command places the name of a FORTRAN library in the object file that it creates. The library name corresponds to the memory-model and floating-point options that you chose on the **FL** command line, or the defaults for options you did not explicitly choose. The linker automatically searches for a library with this name. Because of this, you do not need to give library names on the **LINK** command line unless you want to add the names of other libraries, search for libraries in different locations, or override the use of the library named in the object file. (Table 3.1 in Section 3.3.1, "Floating-Point and Memory-Model Options," shows the library names that **FL** includes in the object file for each combination of memory-model and floating-point options.)

In cases where you have renamed a standard library or you want to link with different libraries, use the *libfiles* field to specify them. You can give the names of one or more libraries, paths, or drives that you want the linker to search, separated by plus signs (+) or spaces. You must add a

backslash (\), or a colon (:) if it is a drive letter, to the end of any path name so the linker can distinguish it from a file name. Any combination of these entries is allowed. See Section 4.5.3 for information about how this specification affects the library-search process.

4.4.1.6 Specifying Linker Options

■ Field

options

In this field, you can give any of the linker options described in Sections 4.6.1 through 4.6.17 of this manual.

You do not have to give any *options* when you run the linker. If you specify *options*, you can put them anywhere on the command line.

■ Examples

```
LINK FUN+TEXT+TABLE+CARE, ,FUNLIST, XLIB.LIB
```

The command line above causes **LINK** to load and link the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ, and search for unresolved references in the library file XLIB.LIB and the default libraries. By default, the executable file produced by **LINK** is named FUN.EXE. **LINK** also produces a map file named FUNLIST.MAP.

```
LINK FUN,,;
```

This command line produces a map file named FUN.MAP, since a comma appears as a placeholder for the *mapfile* specification on the command line.

```
LINK FUN,;
LINK FUN;
```

These command lines do not produce a map file, since commas do not appear as placeholders for the *mapfile* specification.

4.4.2 Using Prompts

If you want to use prompts to specify input to the **LINK** command, start the linker by typing **LINK** at the DOS command level. **LINK** prompts you for the input it needs by displaying the following lines, one at a time:

```
Object Modules [.OBJ]:
Run File [basename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

LINK waits for you to respond to each prompt before printing the next one. Section 4.5.2 gives the rules for specifying file names in response to these prompts.

The responses you give to the **LINK** command prompts correspond to the fields on the **LINK** command line. (See Section 4.4.1 for a discussion of the **LINK** command line.) The following list shows these correspondences:

Prompt	Command-Line Field
"Object Modules"	*objfiles*
"Run File"	*exefile*
"List File"	*mapfile*
"Libraries"	*libfiles*

If a plus sign (+) is the last character that you type on a response line, the prompt appears on the next line, and you can continue typing responses. In this case, the plus sign must appear at the end of a complete file or library name, path name, or drive name.

Default Responses

To select the default response to the current prompt, type a carriage return without giving a file name. The next prompt will appear.

To select default responses to the current prompt and all the remaining prompts, type a semicolon (;) followed immediately by a carriage return. After you enter a semicolon, you cannot respond to any of the remaining prompts for that link session. Use this option to save time when you want to use the default responses. Note, however, that you cannot enter a semicolon in response to the "Object Modules" prompt, because there is no default response for that prompt.

The following list shows the defaults for the other linker prompts:

Prompt	Default
"Run File"	The name of the first object file submitted for the "Object Modules" prompt, with the **.EXE** extension replacing the **.OBJ** extension
"List File"	The special file name **NUL.MAP**, which tells **LINK** *not* to create a map file
"Libraries"	The default libraries encoded in the object module (see Section 4.4.1.5, "Specifying Libraries")

4.4.3 Using a Response File

To operate the linker with a response file, you must set up the response file and then type the following:

LINK *@filename*

Here *filename* gives the name of the response file. This may also be a path name. You can name the response file anything you like.

A response file contains responses to the **LINK** prompts. You may give options at the end of any response or place them on one or more separate lines. The responses must be in the same order as the **LINK** prompts discussed in Section 4.4.2. Each new response must appear on a new line or must begin with a comma; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line.

You can also enter the name of a response file after any **LINK** command prompt or at any position in the **LINK** command line.

LINK treats the input from the response file just as if you had entered it in response to prompts or in a command line. It treats any carriage-return–line-feed combination in the response file the same as if you had pressed the ENTER key in response to a prompt or included a comma in a command line.

You can use options and command characters in the response file in the same way as you would use them in responses you type at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, **LINK** uses the default responses for the executable file and for the remaining prompts.

When you enter the **LINK** command with a response file, each **LINK** prompt is displayed on your screen with the corresponding response from your response file. If the response file does not include a line with a file name, semicolon, or carriage return for each prompt, **LINK** displays the missing prompts and waits for you to enter responses. When you type an acceptable response, **LINK** continues the link session.

■ Example

Assume that the following response file is named FUN.LNK:

```
FUN TEXT TABLE CARE
/PAUSE /MAP
FUNLIST
GRAF.LIB
```

You can type the following command to run **LINK** and tell it to use the responses in FUN.LNK:

```
LINK @FUN.LNK
```

The response file tells **LINK** to load the four object modules FUN, TEXT, TABLE, and CARE. **LINK** produces an executable file named FUN.EXE and a map file named FUNLIST.MAP. The /PAUSE option tells **LINK** to pause before it produces the executable file so that you can swap disks, if necessary. The /MAP option tells **LINK** to include public symbols and addresses in the map file. **LINK** also links any needed routines from the library file GRAF.LIB. See the discussions of the **/PAUSE** and **/MAP** options in Section 4.6.2 and 4.6.5, respectively, for more information about these options.

4.5 Linking FORTRAN Program Files

Sections 4.5.1 through 4.5.3 describe several special considerations that you should keep in mind when you link FORTRAN program files.

4.5.1 The Program Entry Point

After you link a FORTRAN program, program execution in the executable file begins at a procedure named **_main**. The start-up object module in the standard FORTRAN library contains a call to the **_main** function to begin program execution.

4.5.2 Specifying File Names

You can use any combination of uppercase and lowercase letters for the file names you specify on the **LINK** command line or give in response to the **LINK** command prompts. For example, **LINK** considers the following three file names to be equivalent:

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

If you specify file names without extensions, **LINK** uses the following default file-name extensions:

File Type	Default Extension
Object	**.OBJ**
Executable	**.EXE**
Map	**.MAP**
Library	**.LIB**

You can override the default extension for a particular command-line field or prompt by specifying a different extension. To enter a file name that has *no* extension, type the name followed by a period.

- **Examples**

Consider the following two file specifications:

```
ABC.
ABC
```

If you use the first file specification, **LINK** assumes that the file has no extension. If you use the second file specification, **LINK** uses the *default* extension for that prompt.

4.5.3 Specifying Libraries to Be Searched

Object files compiled with the **FL** command contain the name of the FORTRAN library corresponding to the memory-model and floating-point options given on the **FL** command line (or used by default). **LINK** uses this library name to link your files with the appropriate library automatically.

LINK searches for the default library first in the current working directory, then in any directory specified in the **LIB** environment variable.

Since the object file already contains the name of the correct library, you are not required to specify a library on the **LINK** command line or in response to the **LINK** "Libraries" prompt unless you want to do one of the following:

- Add the names of additional libraries to be searched
- Search for libraries in different locations
- Override the use of one or more default libraries

4.5.3.1 Searching Additional Libraries

You can tell **LINK** to search additional libraries by specifying one or more library files on the command line or in response to the "Libraries" prompt. **LINK** searches these libraries *before* it searches default libraries, and it searches these libraries in the order in which you specify them.

If the library name you give includes a path specification, **LINK** searches only that directory for the library.

If you specify only a library name (without a path specification), **LINK** searches in the following locations to find the given library file:

- The current working directory
- Any path specifications or drive names that you give on the command line or type in response to the "Libraries" prompt, in the order in which they appear (see Section 4.5.3.2)
- The locations given by the **LIB** environment variable

LINK automatically supplies the **.LIB** extension if you omit it from a library-file name. If you want to link a library file that has a different extension, be sure to specify the extension.

4.5.3.2 Searching Different Locations for Libraries

You can tell **LINK** to search additional locations for libraries by giving a drive name or path specification in the *libfield* on the command line or in response to the "Libraries" prompt. You can specify up to 16 additional paths. If you give more than 16 paths, **LINK** ignores the additional paths without displaying an error message.

LINK searches for the default libraries in the same order as for libraries given on the command line. See Section 4.5.3.1, "Searching Additional Libraries," for more information.

4.5.3.3 Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the name of a different library instead. You might want to specify a different library name in the following cases:

- If you assigned a "custom" name to a standard library when you ran **SETUP** and you want to be sure that you link with that library.
- If you want to link with a library that supports a different math package than the math package you gave on the **FL** command line (or the default). In this case, you must have compiled with the **/FPc**, **/FPc87**, or **/FPa** option. (See Chapter 8, "Controlling Floating-Point Operations," for a discussion of these options.)

If you specify a new library name on the **LINK** command line, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the library whose name is included in the object file, you must use the **/NOD** option. This option tells **LINK** to ignore the default-library information that is encoded in the FORTRAN object files. Use this option with caution; see the discussion of the **/NOD** option in Section 4.6.8 for more information.

- **Example**

```
LINK

Object Modules [.OBJ]: FUN TEXT TABLE CARE
Run File [FUN.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]: C:\TESTLIB\ NEWLIBV3
```

This example links four object modules to create an executable file named FUN.EXE. **LINK** searches NEWLIBV3.LIB before searching the default libraries to resolve references. To locate NEWLIBV3.LIB and the default libraries, the linker searches the current working directory, then the C:\TESTLIB\ directory, and finally, the locations given by the **LIB** environment variable.

4.6 Using Linker Options

This section explains how to use linker options to specify and control the tasks performed by **LINK**. All options begin with the linker's option character, the forward slash (/).

When you use the **FL** command to invoke the linker, any linker options you specify must appear as part of the **/link** option.

When you use the **LINK** command line to invoke **LINK**, options can appear at the end of the line or after individual fields on the line. However, they must precede the comma that separates each field from the next.

If you respond to the individual prompts for the **LINK** command, you can specify linker options at the end of any response. When you specify more than one option, you can either group the options at the end of a single response or distribute the options among several responses. Every option must begin with the slash character (**/**), even if other options precede it on the same line. Similarly, in a response file, options can appear on a line by themselves or after individual response lines.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so that the linker can determine which option you want. (The minimum legal abbreviation for each option is indicated in the description of the option.)

For example, since several options begin with the letters "NO," abbreviations for those options must be longer than "NO" to be unique. You cannot use "NO" as an abbreviation for the **/NOIGNORECASE** option, since **LINK** cannot tell which of the options beginning with "NO" you intend. The shortest legal abbreviation for this option is **/NOI**.

Abbreviations must be begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

Some linker options take numeric arguments. A numeric argument can be any of the following:

- A decimal number from 0 to 65,535.
- An octal number from 0 to 8#177777. A number is interpreted as octal if it starts with 8#. For example, the number `10` is a decimal number, but the number `8#10` is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to #FFFF. A number is interpreted as hexadecimal if it starts with **#** (with no base specifier before the pound sign). For example, `#10` is a hexadecimal number, equivalent to 16 in decimal.

Differences from FL Options

If you are accustomed to using **FL** options, you should be aware that the linker options work in a slightly different manner. Keep the following differences in mind when you use **LINK** options:

- Linker options can be abbreviated; **FL** options cannot. For example, the linker option **/NOIGNORECASE** can be abbreviated to **/NOI**.
- Case is not significant in linker options, as it is in **FL** options. For example, **/NOI** and **/noi** are equivalent.
- The position of a linker option on the command line is not significant; the option affects all files in the linking process, regardless of where it appears in the **LINK** command line or in the **/link** field of the **FL** command line.

4.6.1 Viewing the Options List (/HE)

■ Option

/HE[LP]

The **/HELP** option causes **LINK** to display a list of the available options on the screen. This gives you a convenient reminder of the available options. Do not give a file name when using the **/HELP** option.

4.6.2 Pausing during Linking (/P)

■ Option

/P[AUSE]

Unless you instruct it otherwise, **LINK** performs the linking session from beginning to end without stopping. The **/PAUSE** option tells **LINK** to pause during the link session before it writes the executable (**.EXE**) file to disk. This option allows you to swap disks before **LINK** writes the executable file.

If you specify the **/PAUSE** option, **LINK** displays the following message before it creates the run file:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* corresponds to the current drive. **LINK** resumes processing when you press the ENTER key.

Note

Do not remove the disk that will receive the list file or the disk used for the temporary file.

If a temporary file is created on the disk you plan to swap, you should press CONTROL-C to terminate the **LINK** session. Rearrange your files so that the temporary file and the executable file can be written to the same disk. Then try linking again.

4.6.3 Displaying Linker Process Information (/I)

- **Option**

/I[NFORMATION]

The **/INFORMATION** option tells the linker to display information about the linking process, including the phase of linking and the names of the object files being linked. This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

Output from this option is sent to the standard error output. You can use the **ERROUT** utility, described in Section 7.5, to redirect output to any file or device.

The following example shows a sample of the linker output when the **/I** and **/MAP** options are specified on the **LINK** command line:

```
**** PASS ONE ****
TEST.OBJ(test.for)
**** LIBRARY SEARCH ****
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
    .
    .
    .
**** ASSIGN ADDRESSES ****
  1 segment "TEST_TEXT" length 122H bytes
  2 segment "_DATA" length 912H bytes
  3 segment "CONST" length 12H bytes
    .
    .
    .
**** PASS TWO ****
TEST.OBJ(test.for)
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
    .
    .
    .
**** WRITING EXECUTABLE ****
```

4.6.4 Packing Executable Files (/E)

- **Option**

/E[XEPACK]

The **/EXEPACK** option directs **LINK** to remove sequences of repeated bytes (typically null characters) and optimize the load-time relocation table before creating the executable file. (The load-time relocation table is a table of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.) Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. However, you cannot use the Symbolic Debug Utility (**SYMDEB**) or the CodeView window-oriented debugger to debug with packed files. The **/EXEPACK** option strips symbolic information from the input file and notifies you of this with a warning message.

The **/EXEPACK** option does not always give a significant saving in disk space, and may sometimes actually increase file size. Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters are usually shorter if packed. If you're not sure whether your program meets these conditions, link it both ways and compare the results.

4.6.5 Listing Public Symbols (/M)

- **Option**

/M[AP]

You can list all public (global) symbols defined in an object file or files by using the **/MAP** option. The **/MAP** option forces **LINK** to create a map file. Using the **/MAP** option with **LINK** is equivalent to using the **/Fm** option with the **FL** command. See Section 3.3.7.5, "Formats for Listings," for a description of the format and contents of a map file.

4.6.6 Including Line Numbers in the Map File (/LI)

- **Option**

/LI[NENUMBERS]

You can include the line numbers and associated addresses of your source program in the map file by using the **/LI** option. Ordinarily the map file does not contain line numbers.

To produce a map file with line numbers, you must give **LINK** an object file (or files) with line-number information. You can use the **/Zd** option with the FORTRAN compiler to include line numbers in the object file. If you give **LINK** an object file without line-number information, the **/LI** option has no effect.

The **/LI** option forces **LINK** to create a map file, even if you did not explicitly tell the linker to create a map file. By default, the file is given the same base name as the executable file, plus the extension **.MAP**. You can override the default name by specifying a new map file on the **FL** or **LINK** command line or in response to the "List File" prompt.

4.6.7 Preserving Case Sensitivity (/NOI)

- **Option**

/NOI[GNORECASE]

By default, **LINK** treats uppercase letters and lowercase letters as equivalent. Thus ABC, abc, and Abc are considered the same name. When you use the **/NOI** option, the linker distinguishes between uppercase letters and lowercase letters, and considers ABC, abc, and Abc to be three separate names.

Note

When you link using the **FL** command, the **/NOI** option is used automatically. If you want to link without using **/NOI**, you must invoke **LINK** directly instead of using **FL**.

Since names in FORTRAN are not case sensitive, this option has minimal importance for FORTRAN programs. However, in some languages, such as C, case is significant. If you plan to link your FORTRAN files with C routines, you may want to use this option.

4.6.8 Ignoring Default Libraries (/NOD)

- **Option**

/NOD[EFAULTLIBRARYSEARCH]

The **/NOD** option tells **LINK** *not* to search any library specified in the object file to resolve external references.

In general, FORTRAN programs do not work correctly without a standard FORTRAN library (that is, one of the libraries built by the **SETUP** program). Thus, if you use the **/NOD** option, you should explicitly specify the name of a standard library.

4.6.9 Controlling Stack Size (/ST)

■ Option

/ST[ACK]:*number*

The **/ST** option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal, or hexadecimal) up to 65,536 (decimal). It represents the size, in bytes, of the stack.

Note

Using the **/ST** option has the same effect as using the **/F** option of the **FL** command.

All compilers and assemblers should provide information in the object modules that tell the linker how to set up the stack. For FORTRAN programs, the default stack size is 2K. The default stack size is set by the start-up routine in the standard FORTRAN library.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

Note

You can also use the **EXEMOD** utility, described in Section 7.3, to change the default stack size for FORTRAN program files by modifying the executable-file header. The format of the executable-file header is discussed in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.

4.6.10 Setting the Maximum Allocation Space (/CP)

■ Option

/CP[ARMAXALLOC]:*number*

The **/CP** option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. The operating system uses this value when allocating space for the program before loading it. The option is useful when you want to execute another program from within your program and you need to reserve space for the executed program.

LINK normally requests the operating system to set the maximum number of paragraphs to 65,535. Since this represents all available memory, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the **/CP** option is used, the operating system allocates no more space than the option specified. This means that any additional space in memory is free for other programs.

Note

The start-up module for FORTRAN 4.0 automatically frees any unneeded memory, so using the **/CP** option is not necessary with FORTRAN.

The *number* can be any integer value in the range 1 to 65,535. If *number* is less than the minimum number of paragraphs needed by the program, **LINK** ignores your request and sets the maximum value equal to the minimum. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program. To free more memory for programs compiled in the medium and large memory models, link with **/CP:1**; this leaves no space for the near heap.

Note

You can change the maximum allocation after linking by using the **EXEMOD** utility, which modifies the executable-file header, as described in Section 7.3. The format of the executable-file header is discussed in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.

4.6.11 Controlling Segments (/SE)

■ **Option**

/SE[GMENTS]:*number*

The **/SE** option controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1 to 1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), the linker avoids having to allocate a large amount of storage space for all programs.

When you set the segment limit higher than 128, the linker allocates more space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting the segment *number* to reflect the actual number of segments in the program.

If the number of segments allocated is too high for the amount of memory **LINK** has available to it, you will see the following error message:

```
segment limit too high
```

Set a lower limit and relink.

4.6.12 Setting the Overlay Interrupt (/O)

■ **Option**

/O[[VERLAYINTERRUPT]]:*number*[]

By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The **/OVERLAYINTERRUPT** option allows the user to select a different interrupt number.

The *number* can be a decimal number from 0 to 255, an octal number from octal 0 to octal 0377, or a hexadecimal number from hexadecimal 0 to hexadecimal FF. Numbers that conflict with DOS interrupts can be used; however, their use is not advised.

In general, you should not use **/OVERLAYINTERRUPT** with FORTRAN routines. The exception to this guideline would be a FORTRAN program that uses overlays and spawns another FORTRAN program using overlays; in this case, each program should use a separate overlay-interrupt number, meaning that at least one of the programs should be compiled with **/OVERLAYINTERRUPT**.

4.6.13 Ordering Segments (/DO)

■ **Option**

/DO[[SSEG]]

The **/DOSSEG** option forces segments to be ordered as follows:

1. All segments with a class name ending in **CODE**
2. All other segments outside **DGROUP**
3. **DGROUP** segments, in the following order:
   a. Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)
   b. Any segments not of class **BEGDATA**, **BSS**, or **STACK**
   c. Segments of class **BSS**
   d. Segments of class **STACK**

FORTRAN programs compiled with Version 4.0 of the Microsoft FORTRAN Compiler always use this segment order by default, so you never need to use the **/DOSSEG** option. See Section 11.2.2.1, "Segments," for a discussion of the segment names used by the FORTRAN compiler.

4.6.14 Controlling Data Loading (/DS)

■ Option

/DS[ALLOCATE]

By default, **LINK** loads all data starting at the low end of the data segment. At run time, the **DS** (data segment) register is set to the lowest possible address to allow the entire data segment to be used.

Use the **/DSALLOCATE** option to tell **LINK** to load all data starting at the high end of the data segment instead. In this case, the **DS** register is set at run time to the lowest data-segment address that contains program data.

The **/DSALLOCATE** option is typically used with the **/HIGH** option, discussed in the next section, to take advantage of unused memory within the data segment. You can allocate any available memory below the area specifically allocated for **DGROUP**, using the same **DS** register.

Warning

Do not use the **/DSALLOCATE** option with FORTRAN programs. It should be used only with assembly-language programs.

4.6.15 Controlling Executable-File Loading (/HI)

■ **Option**

/HI[GH]

The executable file can be placed either as low or as high in memory as possible. Use of the **/HIGH** option causes **LINK** to place the executable file as high as possible in memory. Without the **/HIGH** option, **LINK** places the executable file as low as possible.

Note

Do not use the **/HIGH** option with FORTRAN programs. It should be used only with assembly-language programs.

4.6.16 Preserving Compatibility (/NOG)

■ **Option**

/NOG[ROUPASSOCIATION]

The **/NOG** option causes the linker to ignore group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (Versions 2.02 and earlier) and other Microsoft language compilers.

Note

Do not use the **/NOG** option with FORTRAN programs. It should be used only with assembly-language programs.

4.6.17 Preparing for Debugging (/CO)

■ Option

/CO[DEVIEW]

The **/CO** option is used to prepare for debugging with the CodeView window-oriented debugger provided with Version 4.0 of the Microsoft FORTRAN Compiler. This option tells the linker to prepare a special executable file containing symbolic data and line-number information.

You can run this executable file outside the CodeView debugger; the extra data in the file will be ignored. However, to keep file size to a minimum, use the special-format executable file only for debugging; then you can link a separate version without the **/CO** option after the program is debugged.

If you use the **/CO** option when you link object modules that were compiled without the **/Zi** option, the linker writes only limited public-symbol information to the executable file. **FL** automatically uses the **/CO** option during the linking stage if you use **FL** to compile and link in one step; however, if you link in a separate step (using either the **FL** or the **LINK** command), you must give the **/CO** option yourself.

■ Examples

```
FL /Zi TEST.FOR

FL /c /Zi TEST.FOR
LINK /CO TEST.FOR

FL /c /Zi TEST.FOR
FL TEST.OBJ /link /CO
```

All three of these examples have the same effect: the file TEST.FOR is compiled with the **/Zi** option and linked with the **/CO** option.

4.7 Using Overlays

You can direct **LINK** to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data are never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from disk into memory.

You specify overlays by enclosing them in parentheses in the list of object files that you submit to the linker. Each module in parentheses represents one overlay. For example, you could give the following object-file list in the *objfiles* field of the **LINK** command line:

```
a + (b+c) + (e+f) + g + (i)
```

In this example, the modules `(b+c)`, `(e+f)`, and `(i)` are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the resident part (or root) of your program. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can appear only once in a program.

Note

You can create overlaid versions of programs only if you link with the **LINK** command. The **FL** command cannot create overlaid versions.

The linker replaces calls from the root to an overlay, and calls from an overlay to another overlay with an interrupt (followed by the module identifier and offset). By default, the interrupt number is 63 (3F hexadecimal). You can use the **/OVERLAYINTERRUPT** option of the **LINK** command to change the interrupt number; however, this option is not recommended for FORTRAN programs except in exceptional cases (described in Section 4.6.12).

4.7.1 Restrictions on Overlays

You can overlay only modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. Since long calls are the default in Microsoft FORTRAN programs, this restriction does not matter in most cases. However, calls to subroutines modified with the **NEAR** attribute are short (16-bit) calls. This means that you cannot overlay modules containing **NEAR** subroutines if other modules call those subroutines..

4.7.2 Overlay-Manager Prompts

The overlay manager is part of the FORTRAN run-time library. If you specify overlays during linking, the code for the overlay manager is automatically linked with the other modules of your program.

When the executable file is run, the overlay manager searches for that file whenever another overlay needs to be loaded. The overlay manager first searches for the file in the current directory; then, if it does not find the file, the manager searches the directories listed in the **PATH** environment variable. When it finds the file, the overlay manager extracts the overlay modules specified by the root program. If the overlay manager cannot find an overlay file when needed, it prompts the user to enter the file name.

Even with overlays, the linker produces only *one* **.EXE** file. This file is opened again and again, as long as the overlay manager needs to extract new overlay modules.

For example, assume that an executable program called PAYROLL.EXE, which does not exist in either the current directory or the directories specified by **PATH**, uses overlays. If the user runs it by entering a complete path specification, the overlay manager displays the following message when it attempts to load overlay files:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

The user can then specify the drive or the directory, or both, where PAYROLL.EXE is located. For example, if the file is located in the directory \EMPLOYEE\DATA\ located on Drive B, the user could enter B:\EMPLOYEE\DATA\ or simply \EMPLOYEE\DATA\ if the current drive is B.

If the user later removes the disk in Drive B and the overlay manager needs to access the overlay again, it does not find PAYROLL.EXE, and displays the following message:

```
Please insert diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After the overlay file has been read from the disk, the overlay manager displays the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

4.8 Terminating the LINK Session

To terminate a link session, press CONTROL-C while **LINK** is working or while you are entering responses to **LINK** prompts. If you realize that you entered an incorrect response to a previous prompt, press CONTROL-C to exit **LINK** and begin again. You can use the normal DOS editing keys to correct entries at the current prompt.

4.9 How the Linker Works

LINK performs the following steps to combine object modules and produce a run file:

1. Reads the object modules you submit
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads code and data in the segments
6. Reads all relocation references in object modules
7. Performs fixups
8. Outputs a run file (executable image and relocation information)

The "executable image" contains the code and data that constitute the executable file. The "relocation information" is a list of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

You can control the way **LINK** combines a program's segments by using command-line options with the Microsoft FORTRAN Compiler or Microsoft C Compiler, or by using **SEGMENT** and **GROUP** directives in the Microsoft Macro Assembler (**MASM**). See Section 11.2.2 for a discussion of the segment model for FORTRAN programs and for a listing of class names, alignment types, and combine types.

The following sections explain the process **LINK** uses to concatenate segments and resolve references to items in memory. You do not need to understand this information to use the linker, but it may be helpful for advanced users who want to link FORTRAN routines with assembly routines.

4.9.1 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are **BYTE**, **WORD**, **PARA**, and **PAGE**. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively. The default alignment is **PARA**.

When **LINK** encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is **WORD**, **PARA**, or **PAGE**, **LINK** checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, **LINK** pads the image with extra null bytes.

The Microsoft FORTRAN Compiler automatically assigns alignment types to segments. Table 11.1 in Chapter 11, "Interfaces with Assembly Language and C," shows the alignment types of the segments used by each of the standard memory models.

4.9.2 Frame Number

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the sizes of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory that contains one or more bytes of the segment. A frame number is always a multiple of 16 (a paragraph address). The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For **BYTE** and **WORD** alignments, the offset may be nonzero. The offset is always zero for **PARA** and **PAGE** alignments.

The frame number of a segment can be obtained from the map file created by **LINK** when linking the segment. The frame number is the first five hexadecimal digits of the "Start" address specified for the segment.

4.9.3 Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless **LINK** encounters two or more segments having the same class name. Segments having identical class names belong to the same class type, and are copied as a contiguous block to the executable file.

The FORTRAN compiler automatically assigns class types to segments. Table 11.1 in Chapter 11, "Interfaces with Assembly Language and C," shows the class types of the segments used by each of the standard memory models.

The Microsoft FORTRAN and Pascal Compilers (Versions 3.3 and later) and the Microsoft C Compiler (Versions 3.0 and later) use the segment ordering specified by the **/DOSSEG** linker option. This imposes additional constraints on the segment-loading order. See Section 4.6.13 for a discussion of the **/DOSSEG** option.

4.9.4 Combined Segments

LINK uses combine types to determine whether or not two or more segments sharing the same segment name should be combined into one large segment. The valid combine types are **PUBLIC**, **STACK**, **COMMON**, and **PRIVATE**.

If a segment has combine type **PUBLIC**, **LINK** automatically combines it with any other segments having the same name and belonging to the same class. When **LINK** combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K, **LINK** displays an error message.

If a segment has the combine type **STACK**, **LINK** carries out the same combine operation as for **PUBLIC** segments. The only exception is that **STACK** segments cause **LINK** to copy an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has combine-type **COMMON**, **LINK** automatically combines it with any other segments having the same name and belonging to the same class. When **LINK** combines **COMMON** segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has combine type **PRIVATE** only if no explicit combine type is defined for it in the source file. **LINK** does not combine private segments.

The FORTRAN compiler automatically assigns combine types to segments. Table 11.1 in Chapter 11, "Interfaces with Assembly Language and C," shows the combine types of the segments used by each of the standard memory models.

4.9.5 Groups

Groups allow segments to be addressed relative to the same frame address. When **LINK** encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

Segments in a group do not have to be contiguous, belong to the same class, or have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, **LINK** may place segments that do not belong to the group in the same 64K of memory. Although **LINK** does not explicitly check that all segments in a group fit within 64K of memory, **LINK** is likely to encounter a fixup overflow error if this requirement is not met.

The FORTRAN compiler uses a group called **DGROUP** for data segments. For more information on how the Microsoft FORTRAN Compiler uses groups, see Section 11.2.2.2, "Groups."

4.9.6 Fixups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, **LINK** can "fix up" any unresolved references to labels and variables. To fix up unresolved references, **LINK** computes an appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for the types of references shown in the following list:

Type of Reference	Description
Short	Occurs in **JMP** instructions that attempt to pass control to labeled instructions in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference. **LINK** computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or if the target is more than 128 bytes distant in either direction.
Near self relative	Occurs in instructions that access data relative to the same segment or group. **LINK** computes a 16-bit offset for this reference. It displays an error if the data are not in the same segment or group.
Near segment relative	Occurs in instructions that attempt to access data in a specified segment or group, or relative to a specified segment register. **LINK** computes a 16-bit offset for this reference. It displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.
Long	Occurs in **CALL** instructions that attempt to access an instruction in another segment or group. **LINK** computes a 16-bit frame address and 16-bit offset for this reference. **LINK** displays an error message if the computed offset is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.

The size of the value to be computed depends on the type of reference. If **LINK** discovers an error in the anticipated size of a reference, it displays a fixup overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction in a segment having a different frame address. It can also occur if all segments in a group do not fit within a single 64K block of memory.

Chapter 5
Managing Libraries

5.1 Introduction 147
5.2 Using LIB: An Overview 148
5.3 Running LIB 149
5.3.1 Running LIB with a Command Line 149
5.3.1.1 Specifying the Library File 150
5.3.1.2 Specifying a Page Size 150
5.3.1.3 Giving LIB Commands 151
5.3.1.4 Specifying a Cross-Reference-Listing File 152
5.3.1.5 Specifying an Output Library 153
5.3.2 Using Prompts 155
5.3.2.1 Extending Lines 155
5.3.2.2 Using Default Responses 156
5.3.3 Using a Response File 156
5.3.4 Terminating the Library Session 157
5.4 Managing Libraries with LIB 157
5.4.1 Creating a Library File 158
5.4.2 Changing a Library File 158
5.4.3 Adding Library Modules 159
5.4.4 Deleting Library Modules 159
5.4.5 Replacing Library Modules 159
5.4.6 Extracting Library Modules 160
5.4.7 Moving Library Modules 160
5.4.8 Combining Libraries 160

5.4.9 Creating a Cross-Reference-Listing File 161
5.4.10 Performing Consistency Checks 161
5.4.11 Setting the Library-Page Size 162

5.1 Introduction

The Microsoft Library Manager (**LIB**) is a utility designed to help you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. After you have linked a program with a run-time-library file, that program can call a run-time routine exactly as if the function were included in the program. The call to the run-time routine is resolved by finding that routine in the library file.

Run-time libraries are created by combining separately compiled object files into one library file. Library files are usually identified by their **.LIB** extension, although other extensions are allowed.

In addition to accepting DOS object files and library files, **LIB** can read the contents of 286 XENIX archives and Intel-style libraries and combine their contents with DOS libraries. You can add the contents of a 286 XENIX archive or an Intel-style library to a DOS library by using the add command symbol (+).

Once an object file is incorporated into a library, it becomes an object "module." **LIB** makes a distinction between object files and object modules: an object "file" exists as an independent file, while an object "module" is part of a larger library file. An object file can have a full path name, including a drive designation, directory-path name, and file-name extension (usually **.OBJ**). Object modules have only a name. For example, `B:\RUN\SORT.OBJ` is an object-file name, while `SORT` is the corresponding object-module name.

Using **LIB**, you can create a new library file, add object files to an existing library, delete library modules, replace library modules, and create object files from library modules. **LIB** also lets you combine the contents of two libraries into one library file.

The command syntax is straightforward; you can give **LIB** all the input it requires directly from the command line. Once you have learned how **LIB** works and what input it needs, you can use one of the two alternative methods of invoking **LIB**, described in Sections 5.3.1 and 5.3.2. The alternative methods allow you to enter input in response to prompts instead of having to enter the input on the **LIB** command line.

5.2 Using LIB: An Overview

You can perform a number of library-management functions with **LIB**, including the following tasks:

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Append an object file as a module of a library, or append the contents of a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules

For each library session, **LIB** reads and interprets the user's commands as listed below. It determines whether a new library is being created or an existing library is being examined or modified.

1. **LIB** processes deletion and extraction commands (if any).

   LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

2. **LIB** processes any addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or extraction commands, a new library file is created in the addition stage by copying the original library file.)

As **LIB** carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The linker uses the library index to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. **LIB** produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Therefore, when you terminate **LIB** for any reason, you do not lose your original file. It also means that when you run **LIB**, enough space must be available on your disk for both the original library file and the copy.

When you change a library file, **LIB** lets you specify a different name for the file containing the changes. If you use this option, the modified library is stored under the name you give, and the original, unmodified version is preserved under its own name. If you choose not to give a new name, **LIB** gives the modified file the original library name, but keeps a backup copy of the original library file. This copy has the extension **.BAK** instead of **.LIB**.

5.3 Running LIB

You run **LIB** by typing the **LIB** command on the DOS command line. You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. (This type of file is known as a "response file.")

5.3.1 Running LIB with a Command Line

You can start **LIB** and specify all the input it needs from the command line. In this case, the **LIB** command line has the following form:

LIB *oldlib* [[**/PAGESIZE:***number*]] [[*commands*]][[,[[*listfile*]][[,[[*newlib*]]]]][[;]]

To tell **LIB** to use the default responses for the remaining fields, use a semicolon (;) after any field except the *oldlib* field. The semicolon should be the last character on the command line.

Sections 5.3.1.1 through 5.3.1.5 describe the input that you give in each command-line field.

5.3.1.1 Specifying the Library File

■ Field

oldlib[[;]]

Use the *oldlib* field to give the name of the library file you want. Usually library files are named with the **.LIB** extension. You can omit the **.LIB** extension when you give the library-file name since **LIB** assumes that the file-name extension is **.LIB**. If your library file does not have the **.LIB** extension, be sure to include the extension when you give the library-file name. Otherwise, **LIB** cannot find the file.

Path names are allowed with the library-file name. You can give **LIB** the path name of a library file in another directory or on another disk. There is no default for this field. **LIB** produces an error message if you do not give a file name.

If you give the name of a library file that does not exist, **LIB** displays the following prompt:

```
Library file does not exist. Create?
```

Type `y` to create the library file, or `n` to terminate **LIB**. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

If you type an *oldlib* name and follow it immediately with a semicolon (;), **LIB** performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. **LIB** prints a message only if it finds an invalid object module; no message appears if all modules are intact.

5.3.1.2 Specifying a Page Size

■ Option

[[**/PAGESIZE:***number*]]

You can use this option to specify a page size for the library. See Section 5.4.11, "Setting the Library-Page Size," for more information.

5.3.1.3 Giving LIB Commands

■ Field

[[*commands*]]

In this field, you can type one of the command symbols for manipulating modules (+, −, −+, *, or −*), followed immediately by a module name or an object-file name. You can specify more than one operation in this field, in any order. If you leave this field blank, **LIB** does not make any changes to *oldlib*.

Command Symbol	Meaning
+	The add command symbol. A plus sign makes an object file the last module in the library file. Immediately following the plus sign, give the name of the object file. You can use path names for the object file. **LIB** automatically supplies the **.OBJ** extension, so you can omit the extension from the object-file name. You can also use the plus sign to combine two libraries. When you give a library name following the plus sign, a copy of the contents of the given library is added to the library file being modified. You must include the **.LIB** extension when you give a library-file name. Otherwise, **LIB** uses the default **.OBJ** extension when it looks for the file.
−	The delete command symbol. A minus sign deletes a module from the library file. Immediately following the minus sign, give the name of the module to be deleted. A module name has no path name and no extension.

– +	The replace command symbol. A minus sign followed by a plus sign replaces a module in the library. Following the replacement symbol, give the name of the module to be replaced. Module names have no path names and no extensions. To replace a module, **LIB** deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an **.OBJ** extension and to reside in the current working directory.
*****	The copy command symbol. An asterisk followed by a module name copies a module from the library file into an object file of the same name. The module remains in the library file. When **LIB** copies the module to an object file, it adds the **.OBJ** extension and the drive designation and path name of the current working directory to the module name to form a complete object-file name. You cannot override the **.OBJ** extension, drive designation, or path name given to the object file. However, you can later rename the file or copy it to whatever location you like.
– *	The move command symbol. A minus sign followed by an asterisk moves an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

5.3.1.4 Specifying a Cross-Reference-Listing File

■ Field

[[*listfile*]]

This field allows you to give a file name for a cross-reference-listing file. You can specify a full path name for the listing file to cause it to be created outside your current working directory. You can give the listing file any name and any extension. **LIB** does not supply a default extension if you omit the extension.

A cross-reference-listing file contains the following two lists:

1. An alphabetical list of all public symbols in the library.

   Each symbol name is followed by the name of the module in which it is referenced.

2. A list of the modules in the library.

   Under each module name is an alphabetical listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special file name **NUL**, which tells **LIB** *not* to create a listing file.

5.3.1.5 Specifying an Output Library

■ Field

[[*newlib*]]

In the *newlib* field, you can give the name of a new library file that will have the specified changes. This prompt appears only if you specify changes to the library in the *commands* field. The default is the current library-file name.

If you do not specify a new library-file name, the original, unmodified library is saved in a library file with the same name but with a **.BAK** extension replacing the **.LIB** extension.

■ Examples

```
LIB LANG-+HEAP;
```

The example above uses the replace command symbol (−+) to instruct **LIB** to replace the HEAP module in the library LANG.LIB. **LIB** deletes the HEAP module from the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon at the end of the command line tells **LIB** to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written to the original library file instead of creating a new library file.

```
LIB LANG-HEAP+HEAP;

LIB LANG+HEAP-HEAP;
```

The examples above do the same thing as the first example in this section, but in two separate operations, using the add (+) and delete (−) command symbols. The effect is the same for these examples because delete operations are always carried out before add operations, regardless of the order of the operations in the command line. This order of execution prevents confusion when a new version of a module replaces an old version in the library file.

```
LIB FOR;
```

The example above causes **LIB** to perform a consistency check of the library file FOR.LIB. No other action is performed. **LIB** displays any consistency errors it finds and returns to the operating-system level.

```
LIB LANG,LCROSS.PUB
```

This example tells **LIB** to perform a consistency check of the library file named LANG.LIB and then create a cross-reference-listing file named LCROSS.PUB.

```
LIB FIRST -*STUFF *MORE, ,SECOND
```

The last example instructs **LIB** to move the module STUFF from the library FIRST.LIB to an object file called STUFF.OBJ. The module STUFF is removed from the library in the process. The module MORE is copied from the library to an object file called MORE.OBJ; the module remains in the library. The revised library is called SECOND.LIB. It contains all the modules in FIRST.LIB except STUFF, which was removed by using the move command symbol (−*). The original library, FIRST.LIB, remains unchanged.

5.3.2 Using Prompts

If you want to respond to individual prompts to give input to **LIB**, start **LIB** at the DOS command level by typing **LIB**. **LIB** prompts you for the input it needs by displaying the following four messages, one at a time:

```
Library name:
Operations:
List file:
Output library:
```

LIB waits for you to respond to each prompt, then prints the next prompt.

The responses you give to the **LIB** command prompts correspond to the fields on the **LIB** command line. (See Sections 5.3.1.1 – 5.3.1.5 for a discussion of the **LIB** command line.) The following list shows these correspondences:

Prompt	Command-Line Field
"Library name"	The *oldlib* field and the optional **PAGESIZE:***number* option (see Sections 5.3.1.1 and 5.3.1.2, respectively). If you want to perform a consistency check on the library, type a semicolon (**;**) immediately after the library name.
"Operations"	Any of the commands allowed in the *commands* field (see Section 5.3.1.3).
"List file"	The *listfile* field.
"Output library"	The *newlib* field.

5.3.2.1 Extending Lines

If you have many operations to perform during a library session, use the ampersand command symbol (**&**) to extend the operations line. Give the ampersand symbol after an object-module or object-file name; do not put the ampersand between an operation's symbol and a name.

The ampersand causes **LIB** to repeat the "Operations" prompt, allowing you to type more operations.

5.3.2.2 Using Default Responses

After any entry but the first, use a single semicolon (;) followed immediately by a carriage return to select default responses to the remaining prompts. You can use the semicolon command symbol with the command-line and response-file methods of invoking **LIB**, but it is not necessary since **LIB** supplies the default responses wherever you omit responses.

The following list shows the defaults for **LIB** prompts:

Prompt	Default
"Operations"	No operation; no change to library file.
"List file"	The special file name **NUL**, which tells **LIB** not to create a listing file.
"Output library"	The current library name. This prompt appears only if you specify at least one operation at the "Operations" prompt.

5.3.3 Using a Response File

The command to start **LIB** with a response file has the following form:

LIB *@filename*

The *filename* is the name of a response file. The response-file name can be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

You can also enter the name of a response file at any position in a command line or after any of the linker prompts. The input from the response file will be treated exactly as if it had been entered in command lines or after prompts. A carriage-return–line-feed combination in the response file is treated the same as using the ENTER key in response to a prompt, or using a comma in a command line.

Before you use this method, you must set up a response file containing answers to the **LIB** prompts. This method lets you conduct the library session without typing responses to prompts at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would use responses typed on the keyboard. You can type an ampersand at the end of the response to the "Operations" prompt and continue typing operations on the next line.

When you run **LIB** with a response file, the prompts are displayed with the responses from the response file. If the response file does not contain answers for all the prompts, **LIB** uses the default responses.

- **Example**

```
LIBFOR
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes **LIB** to delete the module named HEAP from the LIBFOR.LIB library file, extract the module FOIBLES and place it in an object file named FOIBLES.OBJ, and append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, **LIB** creates a cross-reference-listing file named CROSSLST.

5.3.4 Terminating the Library Session

You can press CONTROL-C at any time during a library session to terminate the session and return to DOS. If you notice that you have entered an incorrect response at a previous prompt, you should press CONTROL-C to exit **LIB** and begin again. You can use the normal DOS editing keys to correct errors at the current prompt.

5.4 Managing Libraries with LIB

The following sections summarize the library-management tasks you can perform with **LIB**. These tasks include creating and changing library files; adding, deleting, replacing, and extracting library modules; creating cross-reference-listing files; performing consistency checks; and setting the page size.

5.4.1 Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlib* field of the command line or at the "Library name" prompt. **LIB** supplies the **.LIB** extension.

The name of the new library file must not be the name of an existing file. If it is, **LIB** assumes that you want to change the existing file. When you give the name of a library file that does not currently exist, **LIB** displays the following prompt:

```
Library file does not exist. Create?
```

Type `y` to create the file, or `n` to terminate the library session. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See the Section 5.4.11, "Setting the Library-Page Size," for a discussion of this option.

Once you have given the name of the new library file, you can insert object modules into the library by using the add command symbol (+) in the *commands* field of the command line or at the "Operations" prompt. You can also add the contents of another library, if you wish. See Section 5.4.3, "Adding Library Modules," and Section 5.4.8, "Combining Libraries," for a discussion of these options.

5.4.2 Changing a Library File

You can change an existing library file by giving the name of the library file at the "Library name" prompt. All operations you specify in the *oldlib* field of the command line or at the "Operations" prompt are performed on that library.

However, **LIB** lets you keep both the unchanged library file and the newly changed version, if you like. You can do this by giving the name of a new library file in the *newlib* field of the command line or at the "Output library" prompt. The changed library file is stored under the new library-file name, while the original library file remains unchanged.

If you don't give a new file name, the changed version of the library file replaces the original library file. Even in this case, **LIB** saves the original, unchanged library file with the extension **.BAK** instead of **.LIB**. Thus, at the end of the session you have two library files: the changed version with the **.LIB** extension and the original, unchanged version with the **.BAK** extension.

5.4.3 Adding Library Modules

Use the add command symbol (+) in the *commands* field of the command line or at the "Operations" prompt to add an object module to a library. Give the name of the object file to be added, without the **.OBJ** extension, immediately following the plus sign.

LIB strips the drive designation and the extension from the object-file specification, leaving only the base name. This becomes the name of the object module in the library. For example, if the object file `B:\CURSOR` is added to a library file, the name of the corresponding object module is `CURSOR`.

Object modules are always added to the end of a library file.

5.4.4 Deleting Library Modules

Use the delete command symbol (−) in the *commands* field of the command line or at the "Operations" prompt to delete an object module from a library. After the minus sign, give the name of the module to be deleted. A module name does not have a path name or extension; it is simply a name, such as `CURSOR`.

5.4.5 Replacing Library Modules

Use the replace command symbol (−+) in the *commands* field to replace a module in the library. Following the replace command symbol, give the name of the module to be replaced. Remember that module names do not have path names or extensions.

To replace a module, **LIB** deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an **.OBJ** extension and to reside in the current working directory.

5.4.6 Extracting Library Modules

Use the copy command symbol (*) followed by a module name in the *commands* field to extract a module from the library file into an object file of the same name. The module remains in the library file. When **LIB** copies the module to an object file, it adds the **.OBJ** extension and the drive designation and path name of the current working directory to the module name. This forms a complete object-file name. You cannot override the **.OBJ** extension, drive designation, or path name given to the object file, but you can later rename the file or extract it to any location you like.

5.4.7 Moving Library Modules

Use the move command symbol (-*) in the *commands* field to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

5.4.8 Combining Libraries

You can add the contents of a library to another library by using the add command symbol (+) with a library-file name instead of an object-file name in the *commands* field. In the *commands* field of the command line or at the "Operations" prompt, give the add command symbol (+) followed by the name of the library whose contents you wish to add to the library being changed. When you use this option, you must include the **.LIB** extension of the library-file name. Otherwise, **LIB** assumes that the file is an object file and looks for the file with an **.OBJ** extension.

In addition to allowing DOS libraries as input, **LIB** also accepts 286 XENIX archives and Intel-format libraries. Therefore, you can use **LIB** to convert libraries from either of these formats to the Microsoft format.

LIB adds the modules of the library to the end of the library being changed. Note that the added library still exists as an independent library. **LIB** copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name in the *newlib* field of the command line or at the "Output library" prompt. If you omit the "Output library" response, **LIB** saves the combined library under the name of the original library being changed. The original library is saved with the same base name and the extension **.BAK**.

5.4.9 Creating a Cross-Reference-Listing File

Create a cross-reference-listing file by giving a name for the listing file in the *listfile* field of the command line or at the "List file" prompt. If you do not give a listing-file name, **LIB** uses the special file name **NUL**, which means that no listing file is created.

You can give the listing file any name and any extension. To cause the listing file to be created outside your current working directory, you can specify a full path name, including drive designation. **LIB** does not supply a default extension if you omit the extension.

A cross-reference-listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols referenced in that module.

5.4.10 Performing Consistency Checks

When you give only a library name followed by a semicolon in the *oldlib* field of the command line or at the "Library name" prompt, **LIB** performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. It is not usually necessary to perform consistency checks, since **LIB** automatically checks object files for consistency before adding them to the library.

To produce a cross-reference-listing file with a consistency check, invoke **LIB** using a command line. Give the library name followed by a semicolon, then give the name of the listing file. **LIB** performs the consistency check, then creates the cross-reference-listing file.

5.4.11 Setting the Library-Page Size

You can set the library-page size while you are creating a library or change the page size of an existing library by adding a page-size option after the library-file name in the **LIB** command line or after the new library-file name at the "Library name" prompt. The option has the following form:

/PAGESIZE:*number*

The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32,768. The option name can be abbreviated to **/P:***number*.

The page size of a library affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size is 16 bytes for a new library or the current page size for an existing library.

Note

Because of the indexing technique used by **LIB**, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of *pagesize*/2 bytes of storage space is wasted. In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Another consequence of this indexing technique is that the page size determines the maximum possible size of the **.LIB** file. Specifically, this limit is *number* * 65,536. For example, `/P:16` means that the **.LIB** file has to be smaller than 1 megabyte (16 * 65,536 bytes).

Chapter 6

Maintaining Programs with MAKE

6.1 Introduction 165
6.2 Using MAKE: An Overview 166
6.3 Creating a MAKE Description File 166
6.4 Maintaining a Program: an Example 170
6.5 Running MAKE 172
6.6 Using MAKE Options 173
6.7 Using Macro Definitions with MAKE 173
6.7.1 Defining and Specifying Macros 174
6.7.2 Using Macros within Macro Definitions 176
6.7.3 Using Special Macros 177
6.8 Defining Inference Rules 177

6.1 Introduction

The Microsoft Program Maintenance Utility (**MAKE**) helps automate program maintenance. **MAKE** is particularly useful during program development: it can update an executable file automatically whenever changes are made to one of its source or object files. However, **MAKE** is more generally useful: it can update *any* file whenever changes are made to other, related files.

Before you run **MAKE**, you must create a file containing the information that **MAKE** needs in order to run. This type of file is known as a "**MAKE** description file." The following example shows a **MAKE** description file named SAMPLE:

```
#SAMPLE IS THE NAME OF THIS FILE
SAMPLE.EXE: SAMPLE.OBJ
        LINK SAMPLE;
```

This description file has the following characteristics:

- SAMPLE.EXE is the name of the "outfile." The outfile is the file that you want **MAKE** to update.
- SAMPLE.OBJ is the name of an "infile." An infile is a file that must have changed before **MAKE** will update the outfile.
- LINK SAMPLE; tells **MAKE** which command to perform to update the outfile. In this example, **MAKE** updates SAMPLE.EXE (the outfile) whenever SAMPLE.OBJ (the infile) has been changed.

To update SAMPLE, you would type the following command:

```
MAKE SAMPLE
```

MAKE then compares the last-modification dates of SAMPLE.EXE and SAMPLE.OBJ. If the date for SAMPLE.OBJ is more recent than the date for SAMPLE.EXE, **MAKE** carries out the **LINK** command, LINK SAMPLE;, specified in the description file. This **LINK** command links the SAMPLE.OBJ file, so that the corresponding executable file, SAMPLE.EXE, is updated automatically to reflect the changes to SAMPLE.OBJ.

6.2 Using MAKE: An Overview

The general procedure for using **MAKE** is as follows:

1. Create a file in which you give **MAKE** the following information:
   a. The name of each outfile that you want it to update
   b. For each outfile, the infiles that must have changed to cause **MAKE** to update the outfile
   c. The commands that you want **MAKE** to perform when any of the infiles change
2. Run **MAKE**. On the command line, give it the name of the description file you have created. (You can also specify options that affect the way in which **MAKE** runs; see Section 6.6 for a description of these options.)

After you invoke **MAKE**, it compares the last-modification date of the infiles with the last-modification date of the corresponding outfiles. If any infile date is more recent than the outfile date, **MAKE** knows that the infile is more up-to-date than the outfile, so **MAKE** automatically carries out the commands given in the description file. Usually, these instructions update the outfile in some way.

The following sections explain how to create a **MAKE** description file and run **MAKE**.

6.3 Creating a MAKE Description File

Since a **MAKE** description file is just a text file, you can use any text editor to create one. You will usually want to give the **MAKE** description file the same file name as the program it updates (with no extension); however, you can use any valid file name.

A **MAKE** description file consists of one or more description blocks, each with the following general form:

```
[[macrodefinition]]
.
.
.
outfile : infile[[,infile...]][[#[ ]comment]]
[[#[ ]comment]]
    command [[#[ ]comment]]
    [[command]] [[#[ ]comment]]
    .
    .
    .
```

The following list defines the items that can appear in this format:

Item	Meaning
macrodefinition	One or more **MAKE** macro definitions. See Section 6.7 for an explanation of how to use macro definitions in a **MAKE** description file.
outfile	The name of a file that you want **MAKE** to update automatically. A colon must separate this field from the *infile* fields.
infile	The names of any files that the outfile depends on. For example, if the outfile is an executable file, the infiles might be object files; if the outfile is an object file, the infiles might be source files. The line containing the *outfile* and *infile* fields is known as the "dependency line."
command	The name of an executable file (for example, **FL** or **LINK**) or a DOS internal command.

Note

One way to remember the **MAKE** description-file format is to think of it in terms of an "if-then" form: if an *outfile* is out of date with any *infile*, or if an *outfile* does not exist, then do *commands*.

The following sections define the rules for using infile and outfile names, commands, comments, and description blocks in a description file.

Outfiles and Infiles

The *outfile* and *infile* options must be valid file names. If any file is not on the same drive and in the same directory as the description file, you must include a path specification with the file name.

In any description block, you can give any number of infile names, but only one outfile name. At least one space must separate each pair of infile names. If you have more infile names than you can fit on one line, type a backslash (\), press ENTER, and then continue typing names on the next line.

Commands

The *command* option in a description block can be any valid DOS command line, consisting of the base name of an **.EXE**, **.COM**, or **.BAT** file or a DOS internal command. You can give any number of commands, but each must begin on a new line and each must appear immediately after a tab or after at least one space.

MAKE carries out this command only if one or more of the infiles in the block has been changed since the outfile was created or most recently updated.

Comments

The number sign (**#**) is a comment character. **MAKE** ignores all characters that follow the comment character on the same line.

If a comment appears on the same line as the outfile name, it must appear after the infile name(s). If a comment appears on a line where a command is expected, the comment character (**#**) must be the first character on the line; no leading white space is allowed.

Description Blocks

You can give any number of description blocks in a description file. You must make sure, however, that a blank line appears between the last line of one description block and the first line of the next description block.

The order in which you place the description blocks is important. **MAKE** examines each description block in turn and makes its decision to carry out the command in that block based on the last-modification date of the outfile and infile. If a command in a later description block changes a file used in an earlier description block, **MAKE** has no way to return to that earlier description block to update files that depend on the changed files.

■ Example

```
STARTUP.OBJ:  STARTUP.FOR
        FL /c /Fs STARTUP.FOR

PRINT.OBJ:    PRINT.FOR #Comment allowed after infile
#Comment before command must start in first column
     FL /c /Fs PRINT.FOR #Comment allowed after command

PRINT.EXE:    STARTUP.OBJ PRINT.OBJ
        LINK STARTUP+PRINT,PRINT,PRINT;
```

This sample description file tells **MAKE** how to update or create three outfiles: STARTUP.OBJ, PRINT.OBJ, and PRINT.EXE. To update or create STARTUP.OBJ and PRINT.OBJ, **MAKE** compiles the outfiles named STARTUP.FOR and PRINT.FOR, respectively. To update or create PRINT.EXE, **MAKE** will link the object files STARTUP.OBJ and PRINT.OBJ.

Note that the description blocks appear in the order in which the outfiles are updated or created. Thus, **MAKE** updates STARTUP.OBJ and PRINT.OBJ (or creates them, if necessary) before it updates or creates PRINT.EXE. Thus, after **MAKE** is run, any changes to the source files STARTUP.FOR and PRINT.FOR will be reflected in PRINT.EXE.

6.4 Maintaining a Program: an Example

Consider a test program called WORK.EXE that is made from two source files, WORK1.FOR and WORK2.FOR. Both source files use an include file named WORK.INC, and both modules must be linked with a library file named LIBV3.LIB. During development, you often compile and link to create WORK.EXE, but you only recompile the source files that have changed.

The following block descriptions in a **MAKE** description file named WORK allow you to update WORK.EXE automatically:

```
WORK.EXE:    WORK.INC
    FL /c /Zi /Od /Fs WORK1.FOR WORK2.FOR

WORK1.OBJ:   WORK1.FOR
    FL /c /Zi /Od /Fs WORK1.FOR

WORK2.OBJ:   WORK2.FOR
    FL /c /Zi /Od /Fs WORK2.FOR

WORK.EXE:    WORK1.OBJ WORK2.OBJ \LIB\LIBV3.LIB
  FL /FeWORK WORK1.OBJ WORK2.OBJ /link \LIB\LIBV3.LIB /CO
```

Each time you finish debugging the program and editing its source files, start **MAKE** with the following command line:

```
MAKE WORK
```

MAKE carries out the following steps (where each step corresponds to a description block):

1. Checks to see if the include file named WORK.INC has been changed since the last time the linker created WORK.EXE. If so, both of the source files WORK1.FOR and WORK2.FOR must be recompiled. (Notice that the **FL** command does not link automatically because another infile, \LIB\LIBV3.LIB, is given for WORK.EXE in the last description block.) If the include file was not changed, **MAKE** proceeds to Step 2.
2. Checks to see if WORK1.FOR has been changed since the last time the compiler created WORK1.OBJ. If so, it carries out the given **FL** command to recompile WORK1.FOR.

3. Checks WORK2.FOR in the same way it checked WORK1.FOR in Step 2. Note that if only one of the source files has been changed, only that file is recompiled. However, if both source files were recompiled in Step 1, then they are not recompiled again in this step.
4. Checks to find out if either of the object files WORK1.OBJ or WORK2.OBJ, or the library file LIBV3.LIB, has been changed since the last time the modules were linked. If either of the object files was recompiled, or if the library file was changed, **MAKE** relinks the program.

If you run **MAKE** with this description file immediately after you create the source files WORK1.FOR and WORK2.FOR, **MAKE** carries out Steps 2 and 3 to compile these source files (since none of the outfiles exist), then links them in Step 4.

If you invoke **MAKE** again without changing any of the infiles, it skips all of the steps in this procedure.

If you change *one* of the source files WORK1.FOR and WORK2.FOR, **MAKE** recompiles that file and then relinks the program in Step 4.

If you change the library file LIBV3.LIB, but make no other changes, **MAKE** skips Steps 1 through 3, but relinks the program in Step 4 (as specified in the last description block).

6.5 Running MAKE

■ Syntax

MAKE [[*options*]] [[*macrodefinitions*]] *filename*

The following list describes the options you can give on the **MAKE** command line:

Option	Meaning
options	One or more of the **MAKE** options described in Section 6.6.
macrodefinitions	One or more **MAKE** macro definitions. The use of macro definitions is described in Section 6.7.
filename	The name of a **MAKE** description file.

Once you start **MAKE**, it reads the line in each description block that specifies the outfile and infiles and checks the modification dates of those files. If any of the infiles has a modification date later than the outfile's modification date, or if the outfile does not exist, **MAKE** displays the commands specified in the block and then executes the given commands. Otherwise, it skips to the next description block.

If **MAKE** cannot find a file, it displays a message informing you that the file was not found. If the missing file is an outfile, **MAKE** continues running since, in many cases, the missing file will be created by later commands.

If the missing file is an infile or a command file (that is, an executable or batch file), **MAKE** stops running. **MAKE** also stops running and displays an exit code if any command in the description block returns an error, unless a minus sign (−) precedes the command line in the **MAKE** description file.

MAKE executes any commands in the environment in which the **MAKE** command itself is invoked. Thus, you can include environment variables such as **PATH** for the commands specified in the description file.

6.6 Using MAKE Options

The options available with the **MAKE** command have the following effects on how **MAKE** operates:

Option	Action
/D	Displays the last modification date of each file as the file is scanned.
/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the **MAKE** description file. **MAKE** continues executing the rest of the description file despite the errors.
/N	Displays commands in the description file that **MAKE** would execute, but does not execute these commands. This option is useful if you are debugging a **MAKE** description file.
/S	Does not display lines as they are executed.

6.7 Using Macro Definitions with MAKE

Macro definitions let you associate a name with text used in a description file, then use the name instead of the text wherever the text appears in a description file. This feature makes it easier to update a description file when one of the names used in the file changes: when you update a macro definition, the corresponding text is updated wherever the macro appears in the definition file. Therefore, you can change the text used throughout the description file without having to edit every line that uses the particular text.

You might want to use macro definitions to perform operations such as the following:

1. Specifying the base names of source, object, and executable files under development. If the program name changes, you only need to change the base name in the macro definition; then the base name is changed automatically for the source, object, and executable files given in the description file.
2. Specifying the set of default options for a command such as **FL** or **LINK**. If the options change, changing the macro definition changes the options wherever the macro appears in the description file.

6.7.1 Defining and Specifying Macros

The form of a macro definition is

name=*text*

After you define a macro, use the following form to include the macro in the description file:

$(*name*)

Wherever the pattern $(*name*) appears in the description file, that pattern is replaced by *text*. The *name* is converted to uppercase; for example, the names `flags` and `FLAGS` are equivalent. If you define a macro name but leave *text* blank, *text* will be a null string.

For *name*, you can also use any environment variable that is defined in the current environment in a macro definition. For example, if the environment variable **PATH** is defined in the current environment, the value of **PATH** will replace any occurrences of `$(PATH)` in the description file.

You can give macro definitions in either of the following two places:

1. In the **MAKE** description file. Each macro definition must appear on a separate line. Any white space (tab or space characters) between *name* and the equal sign (=) or between the equal sign and *text* is ignored. Any other white space is considered part of *text*.
2. On the **MAKE** command line.

To include white space in a macro definition, enclose the entire definition in double quotation marks (" ").

If the same *name* is defined in more than one place, the following order of precedence applies:

1. Command-line definition
2. Description-file task definition
3. Environment definition

■ Example

Assume the following **MAKE** description file named COMPILE:

```
base=ABC
warn="/W0"

$(base).OBJ:        $(base).FOR
        FL /c /Fs   $(warn) $(base).FOR

$(base).exe:        $(base).obj \lib\libv3.lib
        LINK $(base),$(base),$(base);
```

In this description file, macro definitions are given for the names `base` and `warn`.

The `base` macro defines the base name of the source, object, and executable files being maintained. **MAKE** replaces each occurrence of `$(base)` with the text ABC. If the program name changes, you would only have to replace ABC in the macro definition with the new program name to change the base name of all three files.

The `warn` macro specifies the **/W0** option to the **FL** command, which sets the warning level for that command to 0. As for the base name, if the option name ever changed, you would only need to change the macro definition to ensure that the new option name is used wherever the macro appears.

If you want to override one of the macro values in this description file, you can give a new macro definition on the **MAKE** command line, as shown in the following example:

```
MAKE base=DEF compile
```

This command-line definition of `base` overrides the definition of `base` in the description file. This causes `base` to be replaced with DEF instead of ABC.

If you want to override the warning level of 0 for **FL** (as specified by the `warn` macro in the **MAKE** description file) and use the default warning level of 1, instead, you could run **MAKE** with the following command line:

```
MAKE warn= COMPILE
```

Since you give a blank value for `warn` (note the white space between the equal sign and the **MAKE** description-file name), it will be treated as a null string. Because definition on the command line has higher precedence than the definition in the description file, the `$(warn)` macro becomes a null string. Thus, the **/c** and **/Fs** options already specified for the **FL** command are the only ones used.

6.7.2 Using Macros within Macro Definitions

Macros can be used within macro definitions. For example, you could have the following macro definition in a **MAKE** description file named `PICTURE`:

```
LIBS=$(DLIB)\LIBV3.LIB $(DLIB)\GRAPHICS.LIB
```

You could then run **MAKE** and specify the definition for the macro named `$(DLIB)` on the command line, as shown in the following example:

```
MAKE DLIB=C:\LIB PICTURE
```

In this case, every occurrence of the macro `$(DLIB)` in the description file would be expanded to `C:\LIB`, so the definition of the `LIBS` macro in the description file would be expanded to the following:

```
LIBS=C:\LIB\LIBV3.LIB C:\LIB\GRAPHICS.LIB
```

Be careful to avoid infinitely recursive macros such as the following:

```
A = $(B)
B = $(C)
C = $(A)
```

In the example above, if the macro `$(B)` is undefined, all of these macros will be undefined, as well.

6.7.3 Using Special Macros

MAKE recognizes the following special macro names and automatically substitutes the corresponding text for each:

Name	Value Substituted
$*	Base name of the outfile (without the extension)
$@	Complete outfile name
$**	Complete list of infiles

- **Example**

```
TEST.EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
          LINK $**, $@;
          $*
```

In the **LINK** command in the example above, **$**** represents all of the infiles that correspond to the outfile TEST.EXE, and **$@** specifies the complete name of TEST.EXE as the executable-file name on the **LINK** command line. The final line uses **$*** to specify the base name of TEST.EXE, TEST, as the next command to be carried out. Thus, this example is equivalent to the following:

```
TEST:EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
          LINK MOD1.OBJ MOD2.OBJ MOD3.OBJ, TEST.EXE;
          TEST
```

6.8 Defining Inference Rules

Often, you use **MAKE** to perform updates on one type of file when a file of another type is changed. For example, you often use **MAKE** to update object files when source files change or update executable files when object files change.

MAKE allows you to define rules, known as "inference rules," that allow you to give a single command to convert all files with a given extension to files with a different extension. For example, you can use inference rules to specify a single **FL** command that changes any source file (which has an extension of **.FOR**) to an object file (which has an extension of **.OBJ**). You would not have to include the **FL** command in each block in which you compile a source file.

Inference rules have the following form:

.inextension.outextension :
    command
    [[*command*]]
    .
    .
    .

In this format, *command* specifies one of the commands that you must use in order to convert files with extension *inextension* to files with extension *outextension*. Using the earlier example of converting source files to object files, *inextension* would be **.FOR**, *outextension* would be **.OBJ**, and *command* would be the **FL** command with any appropriate command-line options.

If **MAKE** finds a description block without an explicit command, it looks for an inference rule that matches both the outfile extension and the infile extension. If it finds such a rule, **MAKE** carries out any commands given in the rule.

You can include inference rules in one of two places:

1. In a **MAKE** description file.
2. In a file named **TOOLS.INI**. This file is known as the "tools-initialization file." A line beginning with the tag **[make]** must appear before any dependency rules in **TOOLS.INI.**

MAKE searches for dependency rules in the following order:

1. In the current description file.
2. In the **TOOLS.INI** file. **MAKE** looks for **TOOLS.INI** on the current drive and directory, then searches any directories given in the DOS **PATH** command. If **MAKE** finds **TOOLS.INI**, it looks through the file for a line beginning with the tag **[make]**. It applies any appropriate inference rules following this line.

■ Example

```
.FOR.OBJ:
        FL /Fs $*.FOR

TEST1.OBJ: TEST1.FOR

TEST2.OBJ: TEST2.FOR
        FL TEST2.FOR
```

In the sample description file above, line 1 defines an inference rule that executes the **FL** command on line 2 to create an object file whenever a change is made in the corresponding FORTRAN source file. The file name in the inference rule is specified with the special macro name `$*` so that the rule applies to any base name with the `.FOR` extension.

When **MAKE** encounters the infile names for the outfiles `TEST1.OBJ` and `TEST2.OBJ`, it first looks for commands on the next line. When it does not find any commands, **MAKE** checks for a rule that may apply and finds the rule defined in lines 1 and 2 of the description file. **MAKE** applies the rule, replacing the `$*` macro with `TEST1` when it executes the command, so that the **FL** command becomes

```
FL /Fs TEST1.FOR
```

When **MAKE** reaches the infile name for the `TEST2.OBJ` outfile, it does not search for a dependency rule, since a command is explicitly given for this outfile/infile relationship.

Chapter 7

Using EXEPACK, EXEMOD, SETENV, and ERROUT

7.1 Introduction 183
7.2 The EXEPACK Utility 183
7.3 The EXEMOD Utility 185
7.4 The SETENV Utility 188
7.5 The ERROUT Utility 190

7.1 Introduction

The Microsoft FORTRAN Compiler package includes the following utilities that allow you to modify files and change the environment:

Utility	Function
Microsoft EXE File Compression Utility (**EXEPACK**)	Compresses executable files by removing sequences of repeated characters from the file and by optimizing the relocation table.
Microsoft EXE File Header Utility (**EXEMOD**)	Modifies header information in executable files.
Microsoft Environment Expansion Utility (**SETENV**)	Enlarges the DOS environment table. In IBM PC-DOS Versions 3.1, 3.0, 2.1, and 2.0, **SETENV** allows you to use more and larger environment variables.
Microsoft STDERR Redirection Utility (**ERROUT**)	Redirects standard error output from any command to a given file or device.

The following sections explain how to use the **EXEPACK**, **EXEMOD**, **SETENV**, and **ERROUT** utilities.

7.2 The EXEPACK Utility

The **EXEPACK** utility compresses sequences of identical characters from a specified executable file. It also optimizes the relocation table, whose entries are used to determine where modules are loaded into memory when the program is executed. Using **EXEPACK**, you can reduce the size of some files and decrease the time required to load them.

EXEPACK does not always give a significant saving in disk space, and may sometimes actually increase file size because of an enhanced **.EXE** loader. However, programs that have approximately 500 or more entries in the relocation table and long streams of repeated characters are usually shorter if packed.

The **EXEPACK** program has exactly the same function as the **LINK /EXEPACK** option, except that **EXEPACK** works on files that have already been linked. One use for this utility is to pack the executable files provided with the Microsoft FORTRAN Compiler. Some of the programs are already packed on your distribution disk. If you have floppy disks, you may want to pack all programs in order to make more room on your disks.

The **EXEPACK** command-line format is as follows:

EXEPACK *exefile packedfile*

The *exefile* is the file to be packed and *packedfile* is the name for the packed file. The *packedfile* should have a different name or be on a different drive or directory, since **EXEPACK** will not pack a file onto itself.

Important

Do not try to get around the rule against packing a file onto itself by specifying the same file in a different way. You may be able to fool **EXEPACK**, but the result will be a damaged file. If you want the packed file to replace the original, you should use a separate name for the packed file, then delete the original and rename the packed copy.

When using **EXEPACK** to pack an executable overlay file or a file that calls overlays, the packed file should always be renamed with the original name to avoid the overlay-manager prompt (see Section 4.7.2, "Overlay-Manager Prompts").

Note

Using **EXEPACK** on a file containing symbolic debug information will remove that information from the file.

■ **Example**

```
EXEPACK WORK.EXE WORK.TMP
DEL WORK.EXE
RENAME WORK.TMP WORK.EXE
```

In the example above, the executable file WORK.EXE is packed to a temporary file. The original is then deleted and the new packed version is renamed with the original name.

7.3 The EXEMOD Utility

The **EXEMOD** utility allows you to modify fields in the header of an executable file. To use this utility, you need to understand the conventions used for executable-file headers. They are explained in the *Microsoft MS-DOS Programmer's Reference.*

Some of the options available with **EXEMOD** are the same as **LINK** options, except that they work on files that have already been linked. Unlike the **LINK** options, the **EXEMOD** options require that values be specified as hexadecimal numbers.

To display the current status of the header fields, type the following:

EXEMOD *exefile*

To modify one or more of the fields in the file header, type the following:

EXEMOD *exefile* [[**/H**]] | [[**/STACK** *hexnum*]][[**/MIN** *hexnum*]][[**/MAX** *hexnum*]]

EXEMOD expects the *exefile* to be the name of an existing file with the **.EXE** extension. If the file name is given without an extension, **EXEMOD** appends **.EXE** and searches for that file. If you supply a file with an extension other than **.EXE**, **EXEMOD** displays an error message.

The options in examples are shown with the forward slash (**/**) option designator, but a dash (**-**) may also be used. Options can be given in either uppercase or lowercase, but they cannot be abbreviated. The options and their effects are described in the following list:

Option	Effect
/STACK *hexnum*	Allows you to set the size of the stack for your program by setting the initial **SP** (stack pointer) value to *hexnum*. Here, *hexnum* is a hexadecimal value setting the number of bytes. The minimum allocation value is adjusted upward, if necessary. This option has the same effect as the **LINK /STACK** option, except that it works on files that are already linked.
/MIN *hexnum*	Sets the minimum allocation value (that is, the minimum number of 16-byte paragraphs needed by the program when it is loaded into memory) to *hexnum*. Here, *hexnum* is a hexadecimal value setting the number of paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/MAX *hexnum*	Sets the maximum allocation to *hexnum*, where *hexnum* is a hexadecimal value setting the number of paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value. This option has the same effect as the **LINK /CPARMAXALLOC** option.
/H	Displays the current status of the DOS program header. Its effect is the same as entering **EXEMOD** with an *executablefile* but without options. The **/H** option should not be used with other options.

Note

The **/STACK** option can be used on programs assembled with **MASM** or programs compiled with the Microsoft FORTRAN Compiler, Versions 3.0 and later; the Microsoft Pascal Compiler, Versions 3.3 and later; or the Microsoft C Compiler, Versions 3.0 and later. Use of the **/STACK** option on programs developed with other compilers may cause the programs to fail, or **EXEMOD** may return an error message.

EXEMOD works on packed files. When it recognizes a packed file, it will print the following message:

```
packed file
```

It will then continue to modify the file header.

When packed files are loaded, they are expanded to their unpacked state in memory. If the **EXEMOD /STACK** option is used on a packed file, the value changed is the value that **SP** will have after expansion. If either the **/MIN** or the **/STACK** option is used, the value is corrected as necessary to accommodate unpacking of the modified stack. The **/MAX** option operates as it would for unpacked files.

If the header of a packed file is displayed, the **CS:IP** and **SS:SP** values are displayed as they are after expansion. These values are not the same as the actual values in the header of the packed file.

■ Examples

```
>EXEMOD TEST.EXE

Microsoft (R) EXE File Header Utility  Version 4.00
Copyright (C) Microsoft Corp 1985.  All rights reserved.

TEST.EXE                              (hex)          (dec)

Minimum load size (bytes)              419D          16797
Overlay number                            0              0
Initial CS:IP                   0403:0000
Initial SS:SP                   0000:0000              0
Minimum allocation (para)                 0              0
Maximum allocation (para)              FFFF          65535
Header size (para)                       20             32
Relocation table offset                  1E             30
Relocation entries                        1              1
```

The example above shows how to use **EXEMOD** to display the current file header for file TEST.EXE. The meanings of the header fields are given in the *Microsoft MS-DOS Programmer's Reference*.

```
EXEMOD TEST.EXE /STACK FF /MIN FF /MAX FFF
```

Use the command line above to modify the header for TEST.EXE.

```
>EXEMOD TEST.EXE

Microsoft (R) EXE File Header Utility  Version 4.00
Copyright (C) Microsoft Corp 1985.  All rights reserved.

TEST.EXE                                (hex)          (dec)

Minimum load size (bytes)               528D          20877
Overlay number                             0              0
Initial CS:IP                      0403:0000
Initial SS:SP                      0000:00FF            256
Minimum allocation (para)                FF            256
Maximum allocation (para)               FFF           4095
Header size (para)                       20             32
Relocation table offset                  1E             30
Relocation entries                        1              1
```

The example above shows how you would determine the current status of the header for FILE.EXE after using the command in the previous example to modify the header.

7.4 The SETENV Utility

The **SETENV** utility allows you to allocate more environment space to DOS by modifying a copy of **COMMAND.COM**.

Normally, DOS Versions 2.0 and later allocate 160 bytes (10 paragraphs) for the environment table. This may not be enough if you want to set numerous environment variables using the **SET** or **PATH** command. For example, if you have a hard disk with several levels of subdirectories, a single environment variable might take 40 or 50 characters. Since each character uses 1 byte, you could easily require more than 160 bytes if you want to set several environment variables.

Note

SETENV is guaranteed to work only with IBM PC-DOS Versions 2.0, 2.1, 3.0, and 3.1. **SETENV** may or may not work with other versions of DOS. Moreover, you should not use **SETENV** with versions of DOS later than Version 3.1. Consult your DOS manual for information on how to increase environment size in these later versions.

To enlarge the environment table, you must use **SETENV** to modify a copy of **COMMAND.COM**. Make sure you work on a copy and retain an unmodified version of **COMMAND.COM** for backup.

The command line for modifying the environment table is as follows:

SETENV *filename* [[*environmentsize*]]

Normally *filename* specifies **COMMAND.COM**. It must be a valid, unmodified copy of **COMMAND.COM**, though it could have a different name if you renamed it. The optional *environmentsize* is a decimal number specifying the size in bytes of the new allocation; *environmentsize* must be a number greater than or equal to 160, and less than or equal to 65,520. The specified *environmentsize* is rounded up to the nearest multiple of 16 (the size of a paragraph).

If *environmentsize* is not specified, **SETENV** reports the value that the **COMMAND.COM** file is currently allocating.

After modifying **COMMAND.COM**, you must reboot so that the environment table is set to the new size.

■ Examples

```
>SETENV COMMAND.COM

Microsoft (R) Environment Expansion Utility  Version 2.00
Copyright (C) Microsoft Corp 1985, 1986.  All rights reserved.

command.com: Environment allocation = 160
```

In the example above, no environment size is specified, so **SETENV** reports the current size of the environment table.

```
SETENV COMMAND.COM 605
```

In the example above, an environment size of 605 bytes is requested. Since 605 bytes is not on a paragraph boundary (a multiple of 16), **SETENV** rounds the request up to 608 bytes. **COMMAND.COM** is modified so that it will automatically set an environment table of 608 bytes (38 paragraphs). You must reboot to set the new environment-table size.

7.5 The ERROUT Utility

By default, standard output and standard error output from a DOS program are directed to the terminal. The **ERROUT** utility allows you to execute a given program, command, or batch file and redirect standard output or standard error output to a specified device or file.

The **ERROUT** command-line format is as follows:

ERROUT [[**/f** *stderrfile*]] *command*[[> *stdoutfile*]]

The *command* is the base name of the DOS **.EXE**, **.COM**, or batch file whose error output is redirected.

The **/f** *stderrfile* option is the name of the file or device to which standard error output is redirected. The **f** must be lowercase, and a space must separate it from the *stderrfile*.

The > *stdoutfile* option is the name of the file or device to which standard output is redirected. If this option is used without the **/f** option, both standard error output and standard output are redirected to *stdoutfile*.

If one or the other option is not specified, the corresponding output is directed to the terminal as usual.

■ Examples

```
ERROUT /f PLANERR.DOC PLAN>PLANMSG.DOC
```

The example above causes error output from the PLAN program to be redirected to the file named PLANERR.DOC and standard output from the PLAN program to be redirected to the file named PLANMSG.DOC.

```
ERROUT PLAN>PLANMSG.DOC
```

The example above causes both error output and standard output from the PLAN program to be redirected to the file named PLANMSG.DOC.

Chapter 8
Controlling Floating-Point Operations

8.1 Introduction 193
8.2 Summary of Math Packages 193
8.2.1 The 8087/80287 Package 193
8.2.2 The Emulator Package 194
8.2.3 The Alternate Math Package 194
8.3 Selecting Floating-Point Options (/FP) 195
8.3.1 Library Considerations for Floating-Point Options 200
8.3.1.1 In-Line Instructions or Calls 200
8.3.1.2 Using One Standard Library for Linking 200
8.3.2 Compatibility between Floating-Point Options 203
8.3.3 Using $FLOATCALLS and $NOFLOATCALLS 204
8.4 Using the NO87 Environment Variable 204
8.5 Using Non-IBM®-Compatible Computers 205

8.1 Introduction

This chapter discusses the various ways that you can control how your Microsoft FORTRAN programs handle floating-point math operations. It describes the math packages that you can include in FORTRAN libraries when you run the **SETUP** program, then discusses the **FL** command options for choosing the appropriate library for linking and controlling floating-point instructions.

This chapter also explains how to override floating-point options by changing libraries at link time, and how to control use of an 8087 or 80287 coprocessor through the **NO87** environment variable.

8.2 Summary of Math Packages

The Microsoft FORTRAN Compiler offers a choice of the following three math packages for handling floating-point operations:

1. 8087/80287 (default)
2. Emulator
3. Alternate math

When you run the **SETUP** program, you choose one of these three math packages. **SETUP** includes the math package you choose in the library it builds. Any programs that are linked with that library use the math package included in the library; you must use the appropriate **FL** option to make sure that the library you want is used at link time.

The following descriptions of these math packages are designed to help you choose the appropriate math option for your needs when you build a library using **SETUP**.

8.2.1 The 8087/80287 Package

The 8087/80287 allows you to use an 8087 or 80287 coprocessor to perform floating-point operations. You must have an 8087 or 80287 installed to use this package. This is the default math package that **SETUP** uses if you do not explicitly choose another package.

8.2.2 The Emulator Package

The emulator package uses an 8087 or 80287 coprocessor if one is installed. If no coprocessor is installed, it provides many 8087/80287 functions in software. This package is the best choice if you want to maximize accuracy in program results and if the program will be run on systems with and without coprocessors.

The emulator package can perform basic operations to the same degree of accuracy as an 8087/80287. However, the emulator routines used for transcendental math functions differ slightly from the corresponding 8087/80287 functions, and this difference can cause a slight difference (usually within 2 bits) in the results of these operations when performed with the emulator instead of with an 8087/80287.

Important

When you use an 8087 or 80287 coprocessor or the emulator, interrupt-enable, precision, underflow, and denormalized-operand exceptions are masked by default. The remaining exceptions are unmasked. See Section E.4.2, "Other Run-Time Error Messages," for more information about 8087 floating-point exceptions.

8.2.3 The Alternate Math Package

The alternate math package gives you the smallest and fastest programs you can get without a coprocessor. However, the program results are not as accurate as results given by the emulator package.

The alternate math package uses a subset of the Institute of Electrical and Electronics Engineers, Inc. (IEEE) standard-format numbers; infinities, NANs, and denormal numbers are not used.

8.3 Selecting Floating-Point Options (/FP)

■ Options

/FPa	Generates floating-point calls; selects *x***LIBFORA.LIB**
/FPc	Generates floating-point calls; selects *x***LIBFORE.LIB**
/FPc87	Generates floating-point calls; selects *x***LIBFOR7.LIB**
/FPi	Generates in-line instructions; selects *x***LIBFORE.LIB**
/FPi87	Generates in-line instructions; selects *x***LIBFOR7.LIB** (default)

The **/FP** options of the **FL** command control how a program will handle floating-point operations. You can use only one of these options on the **FL** command line. The option applies to the entire command line, regardless of its position.

Each **/FP** option includes two parts, which specify the following information:

1. How floating-point instructions are included in the program using in-line 8087/80287 instructions or calls to floating-point library functions. The letter **i** indicates in-line instructions; the letters **c** and **a** indicate floating-point calls.
2. Which floating-point package is selected by default when you link.

Based on the **/FP** option and the memory-model option you choose, the **FL** command embeds a library name in the object file that it creates. (See Table 3.1 in Section 3.3.1, "Floating-Point and Memory-Model Options," for a list of the library names used for each combination.) This library is then considered the default library; that is, the linker searches in the standard places for a library with that name. If it finds a library with that name, the linker uses the library to resolve external references in the object file being linked. Otherwise, it displays a message indicating that it could not find the library.

This mechanism allows the linker to link the object file automatically with the appropriate library. However, as explained later in this section and in Section 8.3.1, "Library Considerations for Floating-Point Options," you are allowed to link with a different library in some cases.

Table 8.1 summarizes the **/FP** options and their effects.

Table 8.1

Summary of Floating-Point Options

Option	Method	Advantages	Use of Coprocessor	Libraries Selected
/FPi87	In-line	Smallest and fastest option available with a coprocessor	Requires coprocessor	**LLIBFOR7.LIB** or **MLIBFOR7.LIB**[1]
/FPc87	Calls	Slower than **/FPi87**, but allows changing library at link time	Requires coprocessor	**LLIBFOR7.LIB** or **MLIBFOR7.LIB**[2]
/FPi	In-line	Larger than **/FPi87**, but can work without coprocessor. Most efficient way to get maximum precision without a coprocessor.	Uses coprocessor if present[3]	**LLIBFORE.LIB** or **MLIBFORE.LIB**
/FPc	Calls	Slower than **/FPi**, but allows changing library at link time	Uses coprocessor if present[3]	**LLIBFORE.LIB**[2] or **MLIBFORE.LIB**

Table 8.1 *(continued)*

Option	Method	Advantages	Use of Coprocessor	Libraries Selected
/FPa	Calls	Fastest and smallest option available without coprocessor, but sacrifices some accuracy to speed	Ignores coprocessor	**LLIBFORA.LIB** or **MLIBFORA.LIB**[2]

[1] Can be linked explicitly with **LLIBFORE.LIB** or **MLIBFORE.LIB** at link time. If an emulator library is used, use of the coprocessor must be suppressed by setting **NO87**.

[2] Can be linked explicitly with any library of the right memory model at link time.

[3] Use of the coprocessor can be suppressed by setting **NO87**.

Note

The **/AL** (large) memory-model option is the default. Therefore, if no memory-model option is given on the same **FL** command line, the default library for each floating-point option is **LLIBFOR*x*.LIB** (where *x* is **7**, **E**, or **A**, depending on the math package the library supports). If the **/AM** memory-model option is given, the default library is **MLIBFOR*x*.LIB**.

The following paragraphs discuss the **/FP** options and the advantages and disadvantages of each option.

The /FPi87 Option

The default floating-point option is **/FPi87**, which includes the name of an 8087/80287 library (either **LLIBFOR7.LIB** or **MLIBFOR7.LIB**, depending on the memory model) in the object file. At link time, you can change your mind and link explicitly with an emulator library (either **LLIBFORE.LIB** or **MLIBFORE.LIB**, depending on the memory model). If you use this option and link with an 8087/80287 library, an 8087 or 80287 coprocessor *must* be present at run time; otherwise, the program fails and the following error message is displayed:

```
floating point not loaded
```

If you compile with **FPi87** and link with an emulator library, and if a coprocessor is present at run time, you must set the **NO87** environment variable to suppress the use of the coprocessor. (See Section 8.4 for a description of **NO87**.) If you link with an 8087/80287 library, the **/FPi87** option is the fastest and smallest option available for floating-point operations.

The /FPc87 Option

The **/FPc87** option generates function calls to routines in the 8087/80287 library (**LLIBFOR7.LIB** or **MLIBFOR7.LIB**, depending on the memory model) that perform the corresponding 8087/80287 instructions. As with the **/FPi87** option, you must have an 8087 or 80287 coprocessor installed in order to run programs compiled with this option and linked with an 8087/80287 library. However, the **/FPc87** option gives you more flexibility than the **/FPi87** option. This is because **/FPc87** allows you to change your mind at link time and link with an alternate math library instead of an 8087/80287 or emulator library. See Section 8.3.1, "Library Considerations for Floating-Point Options," for information about changing libraries at link time.

Note

Certain optimizations are not performed when **/FPc87** is used. This may further reduce the efficiency of your code; and, since arithmetic of different precision may result, there may be slight differences in the results.

The /FPi Option

The **/FPi** option generates in-line instructions for an 8087 or 80287 coprocessor and places the name of the emulator library (**LLIBFORE.LIB** or **MLIBFORE.LIB**) in the object file. This option is particularly useful when you do not know in advance whether or not an 8087/80287 coprocessor will be available at run time. If a coprocessor is present at run time, the program uses the coprocessor. If not, the program uses the emulator. If a coprocessor is not present at run time, the **/FPi** option offers the most efficient way to get maximum precision in floating-point results.

The /FPc Option

The **/FPc** option generates floating-point calls to the emulator library and then places the name of the emulator library (**LLIBFORE.LIB** or **MLIBFORE.LIB**, depending on the memory model) in the object file. The **/FPc** option is more flexible than **/FPi**, since it allows you to change your mind at link time and link with an 8087/80287 or alternate math library instead of an emulator library. See Section 8.3.1, "Library Considerations for Floating-Point Options," for information about changing libraries at link time. This option is also recommended if you will be linking with libraries other than the libraries that **SETUP** builds.

The /FPa Option

The **/FPa** option generates floating-point calls and selects the alternate-math library (**LLIBFORA.LIB** or **MLIBFORA.LIB**, depending on the memory model). Calls to this library provide your fastest and smallest option if you do not have an 8087 or 80287 coprocessor. With this option, as with the **/FPc** option, you can change your mind at link time and use an emulator or 8087/80287 library instead.

Note that some expressions may be evaluated at compile time. Such evaluations always use the highest precision possible and are unaffected by the floating-point option you choose.

8.3.1 Library Considerations for Floating-Point Options

Sometimes you may want to use other libraries in addition to the default library for the floating-point option you have chosen on the **FL** command line. For example, you may want to create your own libraries or other collections of subprograms in object-file form and link these libraries at a later time with object files that you have compiled using different **FL** options. The following paragraphs discuss these cases and how to handle them. Although the discussion assumes that you are putting your precompiled object files into libraries, the same considerations apply if you are simply using individual object files.

8.3.1.1 In-Line Instructions or Calls

First, you should decide whether you want to use in-line instructions and compile with the **/FPi87** or **/FPi** option, or floating-point function calls and compile with the **/FPc87**, **/FPc**, or **/FPa** option.

If you choose in-line instructions for your precompiled object files, you cannot use the alternate math package (that is, you cannot link with an *x***LIBFORA.LIB** library). However, you get the best performance from your code on machines that have an 8087 or 80287 coprocessor installed.

If you choose calls, your code is slower, but at link time you can use any standard FORTRAN library—that is, any library created by the **SETUP** program—that supports the memory model you have chosen.

8.3.1.2 Using One Standard Library for Linking

You must also be sure that you use only one standard FORTRAN library when you link. You can control which library is used in one of two ways:

1. At link time, as the *first* name in the list of object files to be linked, give an object file that contains the name of the desired library. For example, if you want to use the alternate math library, you must give the name of an object file compiled using the **/FPa** option. All floating-point calls in this object file refer to the alternate math library.
2. At link time, give the **/NOD** (no default library search) option and then give the name of the library file containing the floating-point package you want to use in the "Libraries" field or in response to the "Libraries" prompt. This library overrides the library names in the object files, and all floating-point calls refer to the named library.

Deciding which standard library to use can become complicated since each library name mentioned in one of the object files being linked is added to the "linker search list" (the list of libraries that the linker searches).

Suppose, for example, that you use the **/FPa** option to create a set of object files and then use the **LIB** utility (described in Chapter 5, "Managing Libraries") to combine these object files into a library. Suppose further that each object file includes a default library name (that is, that you did not use the **/Zl** option to compile). If you want to link this library with an object file that was created using the **/FPc87** option, both **LLIBFOR7.LIB** and **LLIBFORA.LIB** are in the linker search list (assuming you compiled with the default memory-model option). The linker searches libraries on the command line first, so it searches **LLIBFOR7.LIB** before it searches **LLIBFORA.LIB**. Since **LLIBFOR7.LIB** would resolve all external references correctly, this mechanism works correctly.

You can ensure that the standard library of your choice is used for linking by explicitly giving the library name on the **LINK** command line. In this case, **LINK** always searches the library you specify before it searches any libraries named in the object files. However, you must make sure that you specify this library after any of your own libraries on the **LINK** command line. If you don't, and your library contains a different search directive, you may encounter problems.

For example, suppose that the object modules in your library named B were compiled with the **/FPc87** option, so that each module contains a search directive for **LLIBFOR7.LIB**. Suppose further that you are linking with an object file named A that was compiled with the **/FPa** option, so that this object file contains a search directive for **LLIBFORA.LIB**. Finally, suppose that you used the following command line to link your library B with the object file A:

```
LINK A,,,LLIBFOR7+B;
```

In this case, the linker searches libraries in the following order:

1. **LLIBFOR7.LIB** (since it is specified first on the command line)
2. **B** (since it is specified second on the command line)
3. **LLIBFORA.LIB** (since the object module A contains a search directive for this library)
4. **LLIBFOR7.LIB** (since the modules in your library B contain search directives for this library)

The link procedure would proceed as follows:

1. The linker searches **LLIBFOR7.LIB** and resolves all references in object file A to standard run-time routines. This is, presumably, what you intended when you specified this library on the command line.
2. The linker closes **LLIBFOR7.LIB** and searches the next library in the list to satisfy references to routines in your library B. These routines normally contain references to standard run-time routines. Since **LLIBFORA.LIB** is the next library to be searched, this library satisfies the references in B. However, this is not the library you intended to use, since you compiled B with the **/FPc87** option, which uses **LLIBFOR7.LIB** to resolve references to standard run-time routines.

As indicated in this example, you cannot mix libraries in this fashion, and you may get linker errors if you try. Note that if you had specified B+LLIBFOR7.LIB instead of LLIBFOR7.LIB+B on the **LINK** command line, the linker would have searched **LLIBFOR7.LIB** instead of **LLIBFORA.LIB** to resolve standard run-time references in B, and the linking operation would have proceeded correctly.

To avoid this kind of ambiguity and make absolutely sure that you are specifying the correct standard library for linking, use the **/NOD** linker option. This option causes the linker to search only the libraries you specify on the command line.

Perhaps the safest course of all, especially when you are distributing libraries to others, is to compile the object files that make up the library with the **/Zl** option. This option tells the compiler not to include search directives in the object files. Later on, when you link the library with different object files, the standard library used for linking depends only on the floating-point and memory-model options used to compile the later object files. The **/FPc** compiler option is recommended for maximum flexibility in linking with such libraries.

■ Examples

```
FL /c CALC.FOR
LINK CALC+ANOTHER+SUM;
```

In the example above, the source file CALC.FOR is compiled with the default floating-point option, **/FPi87**. **/FPi87** generates in-line instructions and selects the 8087/80287 library (**LLIBFOR7.LIB**, since no floating-point option is given and the large-model library is the default).

```
FL /c /FPa CALC.FOR
FL CALC ANOTHER SUM /link LLIBFORE.LIB /NOD
```

In the example above, CALC.FOR is compiled with the alternate math option (**/FPa**). When the **FL** command is used to link, the **/link** field specifies the **/NOD** option so that the **LLIBFORA.LIB** library (whose name is embedded in the object file **CALC.OBJ**) is not searched. This field gives the name **LIBFORE.LIB** instead, which causes all floating-point calls to refer to the emulator library instead of the alternate math library.

```
FL /c /FPc87 CALC.FOR
FL CALC.OBJ ANOTHER.OBJ SUM.OBJ /link LLIBFORA.LIB /NOD
```

In the example above, CALC.FOR is compiled with the **/FPc87** option, which selects the 8087/80287 library. The **FL** command line used for linking overrides the default library specification by giving the **/NOD** option and the name of the alternate math library (**LLIBFORA.LIB**).

8.3.2 Compatibility between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link more than one object file to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way and that the environment is set up properly to allow the linker to find the required library. See Chapter 2, "Getting Started," for information about choosing floating-point options for the libraries you build with the **SETUP** program; see Chapter 4, "Linking," for more information.

Note

If you are building your own libraries of routines that contain floating-point operations, the **/FPc** floating-point option is recommended for all compilations, as it offers the greatest flexibility.

8.3.3 Using $FLOATCALLS and $NOFLOATCALLS

The **$FLOATCALLS** and **$NOFLOATCALLS** metacommands control whether floating-point operations are processed through calls to library subroutines or by in-line instructions. Use of these metacommands is not recommended. The **/FP** options offer similar control and are more flexible because they select the library to be used as well as the processing method.

However, if you have existing code that contains these metacommands, keep in mind that the **$FLOATCALLS** and **$NOFLOATCALLS** metacommands control only the processing method for floating-point operations. They do not affect the library name that the compiler places in the object file; this is still determined by the **/FP** and **/A** options that you choose on the **FL** command line.

Also note that these metacommands take precedence over the floating-point options you give on the **FL** command line. This may mean, for example, that a source file containing **$NOFLOATCALLS** may result in an object file that contains in-line instructions, even if it was compiled with the **/FPa** option. Such an object file would not link correctly with an alternate math library.

8.4 Using the NO87 Environment Variable

Programs compiled using the **/FPc** or **/FPi** option use an 8087/80287 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting the **NO87** environment variable. (See Section 2.5.1 or your DOS documentation for a discussion of environment variables.)

If **NO87** is currently set to any value when the program is executed, use of the 8087/80287 coprocessor is suppressed. The value of the **NO87** setting is printed on the standard output as a message. The message is only printed if an 8087/80287 is present and suppressed; if no coprocessor is present, no message appears. If you don't want a message to be printed, set **NO87** equal to one or more spaces.

Note that only the presence or absence of the **NO87** definition is important in suppressing use of the coprocessor. The actual value of the **NO87** setting is used only for printing the message.

The **NO87** variable takes effect with any program linked with the emulator library (**LLIBFORE.LIB** or **MLIBFORE.LIB**). It has no effect on programs linked with **LLIBFOR7.LIB**, **MLIBFOR7.LIB**, **MLIBFORA.LIB**, or **LLIBFORA.LIB**.

■ Examples

```
SET NO87=Use of coprocessor suppressed
```

The example above causes the message

```
Use of coprocessor suppressed
```

to appear on the screen when a program that can use an 8087 or 80287 coprocessor is executed.

```
SET NO87=space
```

The example above sets the **NO87** variable to the space character. Use of the coprocessor is still suppressed, but no message is displayed.

```
SET NO87=
```

The example above suppresses the use of the **NO87** variable. Programs that can use an 8087/80287 coprocessor use the coprocessor, if one is present.

8.5 Using Non-IBM®-Compatible Computers

If your computer is not an IBM computer or a closely compatible computer and you want to use an 8087 or 80287 coprocessor, you may have to take special steps to ensure that exceptions are handled correctly. All Microsoft languages that support the 8087 and 80287 coprocessors need to intercept exceptions in order to produce accurate results and detect error conditions properly. Most other languages vendors do not correct or detect these errors.

The exception handler in the emulator and the 8087/80287 libraries (**LLIBFORE.LIB** or **MLIBFORE.LIB**, and **LLIBFOR7.LIB** or **MLIBFOR7.LIB**, respectively) is designed to work without modification on the following computers:

- The IBM PC family of computers
- Computers that are closely compatible with the IBM PC computer, such as the Wang® PC and the AT&T® 6300
- The Texas Instruments® Professional Computer (even though it is not IBM compatible)
- Any machine that uses NMI (nonmaskable interrupts) for 8087 exceptions

If your computer is listed above, or if you are sure that it is completely IBM compatible, you do not need to do anything. If it is not listed, you may need to modify the 8087/80287 libraries.

The distribution disk contains an assembly-language source file, **EMOEM.ASM**, to help make any necessary modifications. Any machine which sends the 8087 exception to an 8259 Priority Interrupt Controller (master or master/slave) can be supported easily by a simple table change to the **EMOEM.ASM** module. The source file contains further instructions on how to modify **EMOEM.ASM** and patch executable files.

Memory Models

Chapter 9
Working with Memory Models

9.1 Introduction 209
9.2 What Is a Memory Model? 211
9.2.1 Code and Data Segments 211
9.2.2 Near, Far, and Huge Addresses 212
9.2.3 The Default Data Segment 214
9.3 FORTRAN Memory Models 215
9.3.1 Limits on Data 216
9.3.1.1 Default-Data-Segment Limits 216
9.3.1.2 Arrays Larger than 64K 217
9.3.1.3 Adjustable-Size and Assumed-Size Arrays 218
9.3.1.4 Common Blocks 219
9.3.1.5 Arguments Passed to Subprograms 220
9.3.1.6 Summary of Data Allocation 221
9.3.2 Limits on Code 222
9.3.2.1 Separate Source Files 222
9.3.2.2 The NEAR Attribute 222
9.4 Selecting and Adjusting the Memory Model 223
9.4.1 Using the Standard Memory Models (/AL, /AH, /AM) 224
9.4.1.1 Large Model 224
9.4.1.2 Huge Model 225
9.4.1.3 Medium Model 226
9.4.2 The NEAR, FAR, and HUGE Attributes 227

9.4.3 The $LARGE and $NOTLARGE Metacommands 231
9.4.4 Using Library Routines with Different Memory Models 232
9.4.5 Setting the Data Threshold (/Gt) 232
9.4.6 Naming Modules and Segments (/NM, /NT) 233

9.1 Introduction

You can gain greater control over how your program uses memory by specifying the program's "memory model." Microsoft FORTRAN provides three standard memory models: the medium, large, and huge models. The characteristics of these models, and strategies for working within their restrictions, are described in Section 9.3, "FORTRAN Memory Models."

You specify a memory model for your programs using the following procedure:

1. When you run the **SETUP** program, you are asked about the memory model you want to use. You can either tell **SETUP** to build a library that supports the default (large) memory model or choose a different memory model. Based on your response, **SETUP** builds support for the selected memory model into the run-time library it creates. If you want to use different memory models for different programs, you should create a separate library for each memory model you plan to use.
2. When you compile a program with the **FL** command, you can give a memory-model option on the command line. This option allows **FL** to link the program automatically with a library that supports that memory model. If you do not specify a memory model, **FL** links with a large-model library by default.

The large memory model is the default for the Microsoft FORTRAN Compiler. The large memory model can accommodate programs that use more than 64K of total code and data. Each module can have up to 64K of code in a unique segment. For programs that require more than 64K of data, the compiler creates multiple data segments as needed by the program. The specific restrictions of the large memory model are explained in Sections 9.3.1 and 9.3.2.

The large memory model can accommodate many of the programs you compile with the Microsoft FORTRAN Compiler. However, if a program does not fit in the large memory model, you must change the memory model. Even if the program will run with the large model, you may be able to improve the program's speed and decrease its size by adjusting the memory model. For example, you can use the large memory model for a program that uses less than 64K of total data, but the program will be more efficient if it is compiled using the medium model.

Note

See Chapter 10, "Improving Compilation and Execution Efficiency," for a description of other ways you can improve program efficiency by choosing libraries during installation, choosing compiler command-line options, and using I/O options within the program.

You can change the memory model by using the **/A***letter* option, as described in Section 9.4.1, "Using the Standard Memory Models." You can also adjust the standard memory models by using any of the following:

- The **NEAR**, **FAR**, and **HUGE** attributes
- The threshold option (**/Gt**)
- The module- and segment-naming options (**/NM** and **/NT**)

Sections 9.4.2, 9.4.5, and 9.4.6, respectively, describe these alternatives. (You can also use the **$LARGE** and **$NOTLARGE** metacommands, described in Section 9.4.3, although this method is not recommended.)

Any large-model library (**LLIBFORE.LIB**, **LLIBFOR7.LIB**, or **LLIBFORA.LIB**) works with either the large or huge model. Medium model requires a different version of the library: **MLIBFORE.LIB**, **MLIBFOR7.LIB**, or **MLIBFORA.LIB**.

If you are already familiar with the addressing conventions of the 8086 family of processors and you understand near, far, and huge addresses, you can skip Section 9.2 and go straight to the description of the FORTRAN memory models in Section 9.3.

If not, Section 9.2 gives an overview of how addressing works on the 8086. This is not intended to be a complete technical description, but simply an outline of the basic concepts needed to understand memory models.

9.2 What Is a Memory Model?

A memory model is a set of predefined rules the compiler follows to map the code and data of the program into segments in memory. The memory model defines how the compiler organizes code and data into segments and what kind of addresses (near, far, or huge) will be used to access the code or data in each segment. A near address is a 16-bit offset that can access a maximum of 64K of memory; a far address or a huge address is a full 32-bit address that can access all of available memory. (See Section 9.2.2 for a discussion of the segmented memory and addresses of the 8086/80286 microprocessor families.)

When you select a memory model, you are telling the compiler that it can make certain assumptions about the program's characteristics and generate code accordingly. For example, when you select the large memory model (the default model for FORTRAN), the compiler expects that the program may have more than 64K of code and more than 64K of data. This means that the compiler must generate far addresses to access both code and data. In medium model, by contrast, the compiler expects only 64K or less of data, allowing it to generate near addresses to access data items.

To understand how memory models work, you must have a basic understanding of the segmented architecture of the 8086 family of processors. The remainder of this section gives an overview of the addressing conventions used on the 8086 family of processors and how they relate to memory models.

9.2.1 Code and Data Segments

The first concept to understand is the broad distinction between a program's *code* and its *data*. A program's code consists of its executable statements in compiled form—the machine instructions that the processor is to carry out. Program data are information used in the course of the program.

When a program is loaded into memory, code and data are placed in separate storage areas. The processor treats the stored code as a sequence of operations to be carried out. A program refers to other code locations by making *calls* to subroutines or functions.

The stored data, on the other hand, do not represent instructions but are simply values, needed by the program, for which an area of memory is reserved. A program refers to data when it uses, for example, variable names, arrays, or common blocks.

All processors make this fundamental distinction between code and data. However, the way in which a program's code and data are stored and accessed depends on the architecture and addressing schemes of the particular processor being used.

On the 8086, code and data occupy separate segments and are accessed through separate segment registers. See Section 9.2.2 for an explanation of the different types of addresses and the use of segment registers on the 8086.

9.2.2 Near, Far, and Huge Addresses

The 8086 processor and its relatives are 16-bit machines. Normally, a machine using 16-bit addresses can access only 64K of memory. The 8086 family of processors uses a special addressing scheme to overcome this limitation.

To extend the amount of memory that can be addressed by a program, physical memory on the 8086 is divided into "segments," each up to 64K long. The starting point of each segment in memory is represented by a 16-bit address. The 8086 reserves four registers to hold segment base addresses: **CS** (code segment), **DS** (data segment), **SS** (stack segment), and **ES** (extra segment).

The segment address, however, points only to the base of the segment. To refer to a particular item within a segment, you must also give the location of the item within the segment. This requires a 16-bit offset address, which gives the address of an item relative to the base of a particular segment. A 16-bit offset is called a "near" address.

A complete address on the 8086, then, requires 32 bits: 16 bits for the segment address and 16 bits for the offset. A full 32-bit address is known as a "far" address.

Wherever possible, it is better for the compiler to generate near addresses, rather than far addresses, to access code and data items. Near addresses are much more efficient than far addresses, because they require less space and take less time to calculate.

Although a complete 8086 address is 32 bits long, the way in which the 8086 uses reserved segment registers makes it possible to access some items with just near addresses. To see how this works, take the simplest case, where a program has one code segment and one data segment. The 8086 has two machine registers that are dedicated to accessing the code and data segments: the **CS** and **DS** registers. When a program is loaded, the **CS** register is set to the address of the code segment, and the **DS** register is set to the address of the data segment. Since these registers are reserved for this purpose, the built-in 8086 instructions assume that the appropriate segment addresses can be found there. Thus, the instructions require only the 16-bit offset of an item within a segment.

Many programs, however, contain more than 64K of code or more than 64K of data. Thus, the addressing scheme becomes somewhat more complicated for larger programs.

When a program has more than 64K of code, it occupies more than one code segment. The Microsoft FORTRAN Compiler places the code from each module (compiled source file) in its own segment. (Each module is therefore restricted to 64K or less, since it must fit in a segment.) All calls to subroutines and functions require the compiler to generate far addresses, since the segment address of the module containing the code must be given, along with its offset. This increases the size of the program and makes it less efficient, but allows it to be larger.

Programs with more than 64K of data occupy more than one data segment. The compiler divides data into different classes (for example, global uninitialized data, constants, and global initialized data) and assigns different classes to different segments. (See Section 11.2.2 for a description of the Microsoft FORTRAN data classes and segments.)

With multiple data segments, some data items are placed in the "default data segment," the data segment addressed by the **DS** register. (See Section 9.2.3 for more information about this data segment.) The compiler only needs to generate a 16-bit offset address to access these items. However, it must generate full 32-bit (far) addresses to access data items outside the default data segment instead of assuming that **DS** holds the appropriate segment value for these items. This makes programs less efficient, but it allows the program to have large amounts of data, a common requirement for FORTRAN programs.

One further complication can arise when a program has very large data items. A program may contain a single data item (array or common block) that exceeds 64K. Normally, the compiler calculates addresses of elements within a data item using 16-bit (near) arithmetic. To do this, it assumes that all elements of the data item lie within the same segment, so the same base address can be used for the address of all elements. When a single data item exceeds 64K, this assumption no longer holds true.

To access elements within a data item that exceeds 64K, the compiler must calculate addresses using 32-bit (far) arithmetic. In Microsoft FORTRAN, a single data item that is larger than 64K is known as a "huge" data item, and the address of the item is a "huge" address. A huge address, like a far address, is a full 32-bit address, but the huge address has the additional implication that 32-bit offsets are required to access individual elements of the data item.

Only data items can have huge addresses. The code for each module is restricted to 64K or less, so it never exceeds one segment.

Huge addresses are even less efficient than far addresses, but they are useful for programs that require very large data items.

9.2.3 The Default Data Segment

Even in programs with multiple data segments, the address of *one* data segment can remain in **DS** throughout program execution. This segment is known as the "default data segment." Items in this segment can be addressed with near addresses.

Local data items (but not formal arguments) that are smaller than the data threshold are placed in this segment, unless a **FAR** or **HUGE** attribute is used to move the item outside the segment. (The data threshold is set with the **/Gt** option, described in Section 9.4.5.) In addition, the default data segment always contains certain internal data, regardless of the data threshold value.

Every program has a default data segment. In a program with only one data segment, the default data segment is the only data segment. See Section 9.3.1.1 for more information about the contents and use of the default data segment.

9.3 FORTRAN Memory Models

This section describes the characteristics of the FORTRAN memory models (medium, large, and huge) and discusses strategies for working within the code and data size limits. Briefly, the standard memory models have the following characteristics:

Model	Characteristics
Large (default)	Total program code and data can each exceed 64K. Each module is given its own code segment (and, thus, is limited to 64K of code). The compiler creates multiple code and data segments as needed. However, formal array arguments are restricted to 64K, unless the argument is explicitly declared with the **HUGE** attribute.
Medium	Total program data are restricted to 64K. Total program code can exceed 64K. Each module is given its own code segment (and, thus, is limited to 64K of code).
Huge	This model has the same characteristics as the large model, but formal array arguments are assumed to exceed 64K.

Many of the code and data size limits apply in all three models. In fact, the only difference between the large and the huge memory model is that the large model assumes that adjustable- and assumed-size arrays are smaller than 64K, while huge model assumes they are larger than 64K. Medium model differs from large model only in its handling of common blocks, and in using near addresses rather than far addresses to pass subroutine arguments. See Section 9.3.1.4 for more information.

Since the three memory models are similar in most respects, the discussion of memory-model characteristics that follows applies to all three models, except where noted.

9.3.1 Limits on Data

The following sections describe how data are allocated in the three FORTRAN memory models, explain the resulting limits on data size, and suggest strategies for working within these limits.

See Table 9.1, "Data Allocation in Large Model," in Section 9.3.1.6, for a quick summary of data allocation.

9.3.1.1 Default-Data-Segment Limits

The Microsoft FORTRAN Compiler stores local variables and arrays smaller than 64K in the default data segment. You can move variables and fixed-size arrays outside the default data segment by explicitly declaring them with the **FAR** or **HUGE** attribute.

In the large and huge models, arrays must also be smaller than the data-threshold value to be stored in the default data segment. The data threshold is a cutoff value the compiler uses in allocating data. Any arrays larger than or the same size as the data threshold are stored in separate segments outside the default data segment and accessed with far addresses. The data threshold is 32,767 bytes by default. You can set it to a different value with the **/Gt** option, described in Section 9.4.5.

The address of the default data segment is always stored in the **DS** register. Items in the default data segment are usually accessed with near (16-bit) addresses, since only an offset from the address in **DS** is required. However, in the large and huge models, 32-bit addresses are used to pass arguments to other routines, unless the formal arguments are declared with the **NEAR** attribute; see Section 9.3.2.2 for more information.

In addition to variables and arrays, the default data segment contains the following:

- The program stack, which is used for arguments passed to subprograms. Normally the stack is 2K, although you can change its size by using the **/F** option with **FL**, the **/STACK** option with **LINK**, or the **EXEMOD** utility.
- Floating-point and character constants, including constants generated by the compiler and I/O routines. These constants may differ in number or value from constants specified in the source program.
- Data allocated or used by the run-time library, including run-time data and internal forms of formats for formatted I/O in medium-model programs. See Section 10.4.2.3 for a discussion of format specifiers.

- Space for file I/O buffers for medium-model programs, and space for dynamic allocation of file-control blocks (FCBs). Whenever an I/O package is linked with a program, FCBs are allocated for the terminal (console), for internal files, and for any additional files that are opened. For every FCB allocated, an associated I/O buffer is allocated, as well; for medium-model programs, these buffers are allocated in the default data segment. (In large-model programs, I/O buffers are allocated outside the default data segment.) If redirection is specified, an additional FCB and an additional I/O buffer are allocated. Note that you can control the size of I/O buffers using the **BLOCKSIZE** option in **OPEN** statements; see Section 10.4.3 for more information about this option.
- Subprogram entry and exit information, if the **/4Yb** option (or the **$DEBUG** metacommand) is in effect.

In the medium memory model, the default data segment also contains all common blocks not explicitly declared with the **HUGE** or **FAR** attribute.

Since the default data segment is limited to 64K, the total space required by the items listed above, plus all local variables and arrays allocated in the default data segment, cannot exceed 64K. If the program violates this restriction, you have the following three options:

1. You can move some data items out of the default data segment by using the **FAR** or **HUGE** attribute. See Section 9.4.2 for more information on these attributes.
2. You can use the **/Gt** option, described in Section 9.4.5, to move all data items larger than a given size out of the default data segment. (This option does not work with medium-model programs.)
3. You can use the **BLOCKSIZE** option in **OPEN** statements in the source program to change the size of the I/O buffers allocated to the units being opened. By default, each unit is allocated a 1024-byte I/O buffer; however, the **BLOCKSIZE** option allows you to specify a different buffer size. This solution is effective only in medium-model programs; that is, the **BLOCKSIZE** option works in large- and huge-model programs, but it does not save you any space in the default data segment.

9.3.1.2 Arrays Larger than 64K

Fixed-size arrays larger than 64K are automatically allocated as many segments as needed outside the default data segment. The compiler generates huge addresses for these arrays since they cross segment boundaries.

The only limit on the number of these huge arrays in the program is available memory. However, no scalar object, including an array element, can span a segment boundary. If possible, the compiler offsets the start of an array in the segment so that this does not occur. However, this cannot be done for arrays (that is, character arrays) whose element size is not a power of 2 and whose length is greater than 128K. Common blocks cannot be adjusted either, since Microsoft FORTRAN does not require different program units to use the same declarations or ordering of variables. If your array or common block declaration tries to allocate a scalar item across a segment boundary, an error will result.

Note

Arrays that are smaller than 64K (65,536 bytes) but larger than 65,521 bytes should be considered to be larger than 64K and declared with the **HUGE** attribute (see Section 9.3.1.3 for more information). Even though such arrays do not exceed the 64K limit, they may be too large for a segment if the segment does not start on a paragraph boundary.

9.3.1.3 Adjustable-Size and Assumed-Size Arrays

Adjustable-size and assumed-size arrays can appear in a FORTRAN program as formal arguments to subprograms. The size of an adjustable-size or assumed-size array is determined at execution time by the size of the array passed as the corresponding actual argument to the subprogram. (See the *Microsoft FORTRAN Compiler Language Reference* for more information on these types of arrays.)

Although the size of such an array is actually determined at execution time, the compiler must decide on an addressing convention at compile time to generate references to the array. To do so, the compiler has to assume either that the array is 64K or smaller (in which case it can generate near or far addresses, depending on where the array is stored) or that the array is larger than 64K (in which case huge addressing is required).

In the large and medium models, the compiler assumes that adjustable-size and assumed-size arrays are 64K or smaller, and therefore generates near or far addresses to access them. If the actual size of an adjustable-size or assumed-size array at execution time is greater than 64K, the program may have undefined results.

To arrange the program so that it will correctly handle adjustable-size and assumed-size arrays larger than 64K, you have the following two choices:

1. You can use the huge memory model (described in Section 9.4.1.2). In huge model, the compiler assumes that adjustable-size and assumed-size arrays are larger than 64K and generates huge addresses to access them. To improve program efficiency in huge model, you have the option of using the **NEAR** or **FAR** attribute with any adjustable-size or assumed-size array whose actual size will always be smaller than 64K.
2. You can specify the **HUGE** attribute when you declare a particular array as a formal argument. This attribute tells the compiler that the actual argument may be (but is not required to be) larger than 64K, causing the compiler to generate huge addresses.

Note that both of the methods mentioned above work in all cases, whether or not the actual size of the adjustable-size or assumed-size array is larger than 64K.

Fixed-size formal array arguments are treated as if they were assumed size, except that a fixed-size formal array argument larger than 64K is implicitly treated as huge. All of the preceding comments about assumed-size arrays apply to fixed-size arrays, too. This is to promote compatibility with earlier versions of FORTRAN that did not support assumed- or adjustable-size arrays but supported the functionality by essentially ignoring the last dimension of an array.

9.3.1.4 Common Blocks

In the large and huge memory models, each common block in the program is allocated as many segments as it needs outside the default data segment. If the common block is larger than 64K, the compiler generates huge addresses. If the common block is 64K or smaller, the compiler generates far addresses.

Some restrictions apply to the variables of a common block larger than 64K. No individual array element or variable in a common block can span a segment boundary. Since the common block is always allocated starting at the beginning of a segment, this restriction means that the boundary between the 65,536th and 65,537th bytes of the array must fall between two variables or between two elements of an array.

If an array in a common block spans a segment boundary and is passed as an actual argument, the corresponding formal argument must be declared with the **HUGE** attribute.

If you have small, frequently accessed common blocks in your program, you may benefit from declaring such blocks with the **NEAR** attribute. This attribute causes the compiler to place the common block in the default data segment, where it can be accessed with more-efficient near addresses. However, this is an option only if you have room in the default data segment; see Section 9.3.1.1, "Default-Data-Segment Limits," for more information.

As an alternative, if you have room in the default data segment, you may be able to use the medium memory model. The medium memory model places all common blocks in the default data segment except the following:

- Blank common blocks explicitly declared with the **FAR** attribute
- Named common blocks that are 64K or smaller

9.3.1.5 Arguments Passed to Subprograms

Normally, FORTRAN arguments are passed by reference. This means that when a call to a subprogram is made, the compiler places the addresses of the arguments on the program stack.

In the large and huge models, the compiler uses far (32-bit) addresses to pass arguments to subprograms, even if the arguments are in the default data segment. However, if a formal argument is declared with the **NEAR** attribute in the **INTERFACE** statement, the compiler uses a near (16-bit) address to pass the actual argument to the subprogram. This means that any actual argument passed to a **NEAR** formal argument must reside in the default data segment.

Since near addresses are more efficient than far addresses, you can improve program efficiency by using the **NEAR** attribute on formal arguments whose actual arguments will always reside in the default segment.

In the medium memory model, the compiler uses near addresses to pass arguments to subprograms, with the assumption that all program data resides in the default data segment. If you want to pass an argument that is outside the default data segment (for example, an array declared with the **FAR** attribute, or any fixed-size array larger than 64K), you must declare the corresponding formal argument with the **FAR** or **HUGE** attribute.

In Microsoft FORTRAN, arguments can also be passed by value. When you use the **VALUE** attribute with an argument, you specify that the argument is to be passed by value instead of by reference. Instead of placing the address of the argument on the stack, the compiler places a copy of the argument's value on the stack.

The stack is part of the default data segment. Default stack size is 2K (you can change the size by using the **/F** option with **FL**, the **/STACK** option with **LINK**, or the **EXEMOD** utility). When passing arguments by value to a subprogram, be sure the arguments' size does not exceed available stack space. Note that passing arrays by value can cause problems.

9.3.1.6 Summary of Data Allocation

Table 9.1 summarizes data allocation in the large memory model. Differences between the large model and the huge and medium models appear in footnotes to the table entries.

Table 9.1
Data Allocation in Large Model

Type of Data	Storage	Type of Address	Restrictions
Local variables and fixed-size arrays smaller than the data threshold[1]	Default data segment	Near[2]	Combined size of all such variables and arrays, plus stack and other data stored in default data segment, cannot exceed 64K.
Arrays smaller than 64K but larger than the data threshold[1]	Separate segments outside default data segment	Far	Available memory
Arrays larger than 64K	As many segments as needed outside default data segment	Huge	Available memory
Common blocks[3]	As many segments as needed outside default data segment	Far if common block is 64K or smaller; huge otherwise	Available memory

[1] The data threshold is 32,767 bytes if no **/GT** option is given or 256 bytes if a **/GT** option is given with no threshold value.

[2] In large and huge models, far addresses are used to pass actual arguments to subroutines (except for formal arguments declared with the **NEAR** attribute); in medium model, near addresses are used (except for formal arguments declared with the **FAR** or **HUGE** attributes).

[3] In medium model, blank common blocks are allocated in the default data segment and restricted to 64K or less unless specifically declared with the **FAR** or **HUGE** attribute. Named common blocks are allocated outside the default data segment if they are larger than 64K.

One feature of the large memory model does not appear in the above table. In large model, the compiler assumes that all formal array arguments are 64K or smaller, unless specifically declared with the **HUGE** attribute. This is also true of the medium model. In the huge model, the compiler assumes that the adjustable-size and assumed-size arrays are larger than 64K, unless specifically declared with the **NEAR** or **FAR** attribute.

9.3.2 Limits on Code

The Microsoft FORTRAN Compiler places the code from each module in its own segment. (A "module" is a compiled source file.) Since the maximum size of a segment is 64K, the compiled code in each module must not exceed 64K. You can determine how much code each program module contains by looking at a source listing or map-file listing.

You can combine the code from two or more modules into one segment by using the **/NT** option, described in Section 9.4.6, "Naming Modules and Segments." Modules with identical text-segment names are loaded into the same segment. Note that the 64K segment limit must still be observed.

The following sections describe some strategies you can use to minimize the size of your program code and make it more efficient. (See Chapter 10, "Improving Compilation and Execution Efficiency," for descriptions of other strategies.)

9.3.2.1 Separate Source Files

If a module exceeds the 64K code-size limit, you must reduce its size by breaking it down into two or more source files. A good practice in developing a large program is to break the source program into subroutines and functions and compile related groups of them separately. Compiling these pieces separately has no effect on the final program size, although it may increase the total size of the object files. This practice has the added benefit of making the source program easier to understand and maintain.

9.3.2.2 The NEAR Attribute

References to code in FORTRAN programs are far (32 bits long), because they must provide both the segment address and the offset of the code item within the segment. If the program uses some subprograms very heavily, you may be able to increase program efficiency by using the **NEAR** attribute.

When applied to a subprogram, the **NEAR** attribute specifies that the subprogram code resides in the same segment as the calling routine. Since the segment address remains constant, the compiler can generate a near (16-bit) address to call the subprogram.

To use the **NEAR** attribute on a subprogram, you must make sure that the subprogram actually appears in the correct segment. You can do this by placing the subprogram in the same source file as the calling routine, thus ensuring that they will be compiled into a single module.

If you want to keep the routines in separate source files, you can use the **/NT** option, described in Section 9.4.6, "Naming Modules and Segments," to set the text-segment names for both modules. If you give both modules the same text-segment name, they will be loaded into the same segment.

See Section 9.3.2.2 of this manual, and the *Microsoft FORTRAN Compiler Language Reference*, for more information on the **NEAR** attribute.

9.4 Selecting and Adjusting the Memory Model

This section describes the methods you can use to select and adjust memory models. You can choose one of the three standard FORTRAN memory models by giving one of the **/A** options on the **FL** command line; these options are described in Section 9.4.1.

When you use the standard memory models, the **FL** command handles library support for you. It automatically links with a library corresponding to the memory model you specify on the command line, provided that you already created this library when you ran the **SETUP** program. Unless you specify otherwise, **FL** uses a large-model library for linking; this is also the default for the library created by the **SETUP** program if you do not choose a memory model during installation. The large and huge models use the same library; the medium model has its own library.

The advantage of using standard models for your program is simplicity. For the standard models, memory management is specified by compiler options and does not require the use of extended keywords. This is an important consideration if you are writing code to be ported to other systems, particularly systems that do not use segmented architecture. (Note that the threshold option, **/Gt**, and the text-segment-naming option, **/NT**, are also portable ways to control allocation.)

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise large-model program with one assumed-size array that may exceed 64K, it is to your advantage to declare the single large array with the **HUGE** attribute, rather than switching to the huge model and forcing *all* assumed- and adjustable-size arrays to be considered huge.

Sections 9.4.2 through 9.4.6 describe ways to adjust allocation for one or more items without changing the entire memory model. The methods described in these sections give you greater control over the program's structure and performance. However, to understand the effects of using the options and attributes described in these sections and to use them safely, you must have a thorough understanding of FORTRAN memory models and the 8086 architecture.

9.4.1 Using the Standard Memory Models (/AL, /AH, /AM)

■ Options

/AL	Large model (default; corresponds to default library generated by **SETUP** program)
/AH	Huge model
/AM	Medium model

The **/AL**, **/AH**, or **/AM** option selects a memory model. You can use only one memory-model option on the command line. The option applies to all source files on the command line, and has no effect on object files given on the command line. If you compile separate source files of a program at different times, you must specify the same memory model for all of them.

9.4.1.1 Large Model

The **/AL** option tells the compiler to use the large memory model. The large memory model is the default, so you do not have to give this option explicitly. The large-model option allows the compiler to create multiple segments as needed for both code and data.

The default in large-model programs is that both code and data items are accessed with far addresses. You can override the default by using the **NEAR** or **HUGE** attribute for data and the **NEAR** attribute for code.

The **/AL** option causes the name of the large-model standard FORTRAN library (**LLIBFORE.LIB**, **LLIBFOR7.LIB**, or **LLIBFORA.LIB**, depending on the **/FP** option you specified on the **FL** command line) to be placed in each object file created. This allows the linker to use the required library automatically, provided that the library exists (that is, that you built it using **SETUP**).

9.4.1.2 Huge Model

The **/AH** option tells the compiler to use the huge memory model.

Huge model differs from large model only in the treatment of adjustable- and assumed-size arrays. In huge model, the compiler assumes that such arrays are larger than 64K and generates huge addresses to access them; in large model such arrays are assumed to be smaller than 64K unless explicitly declared with the **HUGE** attribute.

These huge addresses make huge model less efficient than large model. However, huge model is useful when you want most or all adjustable-size and assumed-size arrays in the program to handle arrays larger than 64K correctly.

When using the huge model, you can use the **NEAR** or **FAR** attribute with any formal array argument whose actual size is always smaller than 64K. This attribute tells the compiler to generate near or far addresses for the array; these addresses are more effective than huge addresses. See Section 9.4.2, "The NEAR, FAR, and HUGE Attributes," and Section 2.6.6 and 2.6.4 of the *Microsoft FORTRAN Compiler Language Reference* for more information about the **NEAR** and **FAR** attributes.

As an alternative to using the huge model, you can use the large model, specifying the **HUGE** attribute for any adjustable-size or assumed-size array whose actual size may exceed 64K. See Section 9.4.2 of this manual and Section 2.6.5 of the *Microsoft FORTRAN Compiler Language Reference* for more information on the **HUGE** attribute.

The same libraries are used for huge model as for large model, so the **/AH** option places the name of the appropriate large-model FORTRAN library (**LLIBFORE.LIB**, **LLIBFOR7.LIB**, or **LLIBFORA.LIB**) in each object file created.

9.4.1.3 Medium Model

The **/AM** option tells the compiler to use the medium memory model.

Medium model differs from large model in two respects: common blocks are allocated differently, and arguments to subprograms are passed differently.

In medium model, blank common blocks not explicitly declared with the **FAR** or **HUGE** attribute and named common blocks that are 64K or smaller are placed in the default data segment. This allocation method allows the compiler to generate near addresses to access these common blocks. As a result, medium-model programs are usually more efficient in terms of speed and space than large- and huge-model programs. Common blocks larger than 64K are placed outside the default data segment and must be accessed with huge addresses.

The argument-passing convention used in medium model is also smaller and faster than in the large and huge models. In medium model, the compiler passes actual arguments to subprograms using near addresses (instead of the far addresses used in large and huge models). This is possible because the compiler assumes that all data reside in the default data segment. If you want to pass data from outside the default data segment to a subprogram in medium model, you must remember to declare the corresponding formal argument in the subprogram with the **FAR** or **HUGE** attribute.

Even if you cannot use the medium model for the program, you can force the compiler to use more efficient near addresses for passing arguments by declaring formal arguments with the **NEAR** attribute. You must be sure, however, that the actual arguments to be passed to the near formal arguments are located in the default data segment.

Important

Input and output on far and huge items are not allowed in medium model. However, you can copy these items to a temporary item in the default data segment to perform I/O on them.

The default data segment is restricted to 64K total, and it contains other data in addition to common blocks. This restriction means that a medium-model program cannot have common blocks larger than 64K, and the total size of the common blocks smaller than 64K must be less than 64K. (See Section 9.3.1.1, "Default-Data-Segment Limits," for a description of the contents of the default data segment.)

In practice, few FORTRAN programs meet this restriction. You may still be able to use the medium model, however, by specifying the **FAR** and **HUGE** attributes in the program. For example, if the program contains large, infrequently accessed common blocks, you can use the **FAR** or **HUGE** attribute to move these blocks out of the default data segment. Smaller common blocks that are accessed more frequently can remain in the default data segment.

You can also use the **FAR** and **HUGE** attributes to move other data items, such as arrays, out of the default data segment. This creates more room in the segment for common blocks.

As an alternative to using the medium memory model, you can use the **NEAR** attribute with specific common blocks to cause them to be placed in the default data segment. This can improve program efficiency when you have one or more small, heavily used common blocks.

See Section 9.4.2 of this manual and Sections 2.6.4 – 2.6.6 of the *Microsoft FORTRAN Compiler Language Reference* for more information about the **NEAR**, **FAR**, and **HUGE** attributes.

The name of the appropriate medium-model library (this is either **MLIBFORE.LIB**, **MLIBFOR7.LIB**, or **MLIBFORA.LIB**, depending on the **/FP** option you specified on the **FL** command line) is placed in each object file created, provided that the library exists (that is, that you built it using **SETUP**).

9.4.2 The NEAR, FAR, and HUGE Attributes

The **NEAR**, **FAR**, and **HUGE** attributes are keywords that can be used in a FORTRAN program to override default addressing conventions. The *Microsoft FORTRAN Compiler Language Reference* describes the syntax of these attributes and outlines restrictions on their use. This section describes the effects of these attributes on code and data items in the three FORTRAN memory models.

These special attributes give you more flexibility than just selecting one of the standard memory models. For example, you may be able to avoid switching from the large to the huge memory model if you use the **HUGE** attribute on any adjustable- or assumed-size arrays that may exceed 64K. On the other hand, even if the program requires the huge model, you can improve program efficiency by identifying small, frequently accessed items that could benefit from being placed in the default data segment.

The **FAR** attribute is also useful for programs that have more data than can fit in the default data segment. You can declare less frequently accessed items with the **FAR** attribute to move them out of the default data segment, leaving room for the more heavily used items. You can also use the **/Gt** option, described in Section 9.4.5, to move some data items out of the default data segment.

Keep in mind, however, that the **NEAR**, **FAR**, and **HUGE** attributes are extensions to the FORTRAN language. They are meaningful only on processors, such as the 8086, that have a segmented architecture. If portable code is a high priority, you should not use these attributes. The standard memory-model options, along with the **/Gt** option, give you a way to alter the memory model in a portable way; since these options are given at compile time, they do not affect the source code.

The discussion of each memory model in the preceding sections suggests strategies for using the **NEAR**, **FAR**, and **HUGE** attributes with each model. Tables 9.2 and 9.3 summarize the effects of each attribute on items in each of the three standard memory models.

Table 9.2

Effects of NEAR Attribute

Item	Large Memory Model	Huge Memory Model	Medium Memory Model
Variables and fixed-size arrays	No effect for items smaller than data threshold value. Arrays larger than threshold value but smaller than 64K are placed in default data segment.[1]	Same as large model	No effect
Formal arguments	Actual arguments are passed with near addresses and must be in default data segment.	Same as large model	No effect
Common blocks	Placed in default data segment; near addresses	Same as large model	No effect
Subprograms	Near calls	Near calls	Near calls
Assumed- and adjustable-size arrays	Actual arguments are passed with near addresses and must be in default data segment.	Same as large model	No effect

[1] The default threshold value is 32,767 bytes if no **/Gt** option is given, or 256 bytes if the **/Gt** option is given with no threshold value.

Table 9.3

Effects of FAR and HUGE Attributes

Item	Large Memory Model	Huge Memory Model	Medium Memory Model
Variables and fixed-size arrays	If variable or array is smaller than 64K, it is moved out of the default data segment and accessed with far addresses. If array is larger than 64K, the **FAR** or **HUGE** attribute has no effect and huge addresses are generated.	Same as large model	Same as large model
Formal arguments	**FAR** has no effect. If **HUGE** is applied to an array, huge addresses are generated.	If **FAR** is used, far addresses are generated instead of huge addresses. **HUGE** has no effect.	Actual arguments are passed using far or huge addresses.
Common blocks	No effect	No effect	Common block moved out of default data segment. Far addresses generated if block is 64K or smaller; huge addresses generated otherwise.
Subprograms[1]	No effect	No effect	No effect
Assumed- and adjustable-size arrays	**FAR** has no effect. Huge addresses are generated with **HUGE** attribute.	If **FAR** is used, far addresses are generated instead of huge addresses. **HUGE** has no effect.	Same as large model

[1] The **HUGE** attribute cannot be applied to subprograms.

9.4.3 The $LARGE and $NOTLARGE Metacommands

The **$LARGE** and **$NOTLARGE** metacommands are related to the **NEAR**, **FAR**, and **HUGE** attributes, although their meanings are slightly different. Use of the **$LARGE** and **$NOTLARGE** metacommands is not recommended; use of the attributes is preferred.

Note

The term "large" can be confusing since it is used both for the **$LARGE** metacommand and for the large memory model. The **$LARGE** metacommand is not related to the large memory model; instead, the size defined by the **$LARGE** metacommand corresponds to the huge memory model (for formal arguments to subprograms) or the **HUGE** attribute (for fixed-size arrays). The **$LARGE** and **$NOTLARGE** metacommands are retained for compatibility with previous versions.

Fixed-size arrays declared while the **$LARGE** metacommand is in effect are treated the same as arrays declared with the **HUGE** attribute: arrays are placed outside the default data segment and must be accessed with far addresses for arrays 64K or smaller or huge addresses for arrays larger than 64K. (In the huge memory model, fixed-size arrays can be accessed with near addresses if they are smaller than a segment and smaller than the specified threshold value.) In the medium and large memory models, declaring an array with the **$NOTLARGE** metacommand (either explicitly or by default) has no effect on the array's allocation. This is also true for fixed-size arrays in huge model.

9.4.4 Using Library Routines with Different Memory Models

The standard libraries built by the **SETUP** program are designed to correspond to the three standard memory models. These libraries impose the following restrictions on FORTRAN programs:

- Huge format specifications and huge arrays as internal files cannot be used in any memory model.
- In medium-model programs, all items involved in I/O operations must be near. These items include items referenced in I/O lists, format specifications, and internal files.
- In medium-model programs, far formats are illegal.

9.4.5 Setting the Data Threshold (/Gt)

■ Option

/Gt[*number*]

The **/Gt** ("Threshold") option sets the data threshold. The data threshold is a cutoff value the compiler uses in allocating data. In the large and huge models, each array or variable larger than or the same size as the threshold value *number*, but smaller than 64K, is stored in a new data segment outside the default data segment and accessed with a far address.

If no **/Gt** option is specified, the threshold value is 32,767 bytes by default. If a **/Gt** option is specified without *number*, the default threshold value is 256 bytes. If *number* is specified, it must follow the **/Gt** option immediately, without intervening spaces.

/Gt has no effect on medium-model programs.

Decreasing the threshold value is useful when you want to move data out of the default data segment without declaring them with the **FAR** attribute. Increasing the threshold value allows you to store arrays larger than 32,767 bytes in the default data segment (provided you have room for them). You can also accomplish this by declaring such arrays with the **NEAR** attribute.

9.4.6 Naming Modules and Segments (/NM, /NT)

- **Options**

/NM*module*
/NT*textsegment*

The **/NM** and **/NT** options let you override the default naming conventions used by the FORTRAN compiler and supply your own names for modules and text segments. The name used with the option can be any combination of letters and digits.

See Section 11.2.2, "The Microsoft FORTRAN Segment Model," for complete information on the segment, group, and class names used by the Microsoft FORTRAN Compiler.

"Module" is another name for an object file created by the Microsoft FORTRAN Compiler. Every module has a name. The compiler uses the name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the **/NM** (for "Name Module") option.

The **/NT** ("Name Text") option sets the name of the text segment, in each module being compiled, to a given name. ("Text" is simply another term for "code"; therefore, a text segment is a code segment.)

The linker uses segment names to define the order in which the segments of the program appear in memory when loaded for execution. See Section 4.9, "How the Linker Works," for more information. The segments in the group named **DGROUP** are an exception; see Section 11.2.2, "The Microsoft FORTRAN Segment Model," for information on **DGROUP**.

Segments with the same name are loaded into the same physical segment in memory. For example, you can use the **/NT** option to give two different modules the same text-segment name, thus ensuring that they will be loaded into the same segment in memory. This is useful when you want to use the **NEAR** attribute with a subprogram.

Text- and data-segment names are normally created by the FORTRAN compiler. In all three of the FORTRAN memory models, the compiler places the code from each module in a separate segment with a distinct name, formed by using the module base name along with the suffix **_TEXT**.

The default data segment is named **_DATA**. The compiler places data that are stored outside the default data segment (huge arrays, arrays larger than the threshold value, items declared with the **FAR** or **HUGE** attribute, and common blocks in large and huge model) in private segments with unique names. (See Section 11.2.2.1, "Segments," for more information on the names.)

Chapter 10

Improving Compilation and Execution Efficiency

10.1 Introduction 237

10.2 Removing Error-Message Text during SETUP 237

10.3 Compiling and Linking Strategies 237

10.3.1 Using the Debug (/4Yb) and Integer-Size (/4I) Options 237

10.3.2 Using 8087/80287 Math Options 238

10.3.3 Linking Version 4.0 and Version 3.3 Modules 239

10.3.4 Using Overlays 239

10.4 Coding Strategies 239

10.4.1 Using Consistent File-Access and Format Types 239

10.4.2 Specifying Edit Lists 240

10.4.2.1 Avoiding Left Tabbing 240

10.4.2.2 Using Formatted or List-Directed I/O 240

10.4.2.3 Using Character Variables as Format Specifiers 240

10.4.3 Using BLOCKSIZE 241

10.4.4 Using Integer and Real Variables 242

10.4.5 Arrays and EQUIVALENCE Statements 242

10.1 Introduction

This chapter offers some ideas for reducing the total size of the executable files that you create with the Microsoft FORTRAN Compiler. Although some of these ideas apply only to a particular class of programs, this chapter may give you ideas for coding programs in a more efficient way. Where applicable, this chapter also discusses program choices that affect compiler efficiency and programs' data size as well as programs' code size.

10.2 Removing Error-Message Text during SETUP

SETUP gives you two options for dealing with error messages: to build a library that displays the entire error message (the default) or a library that displays only the *number* of the error message, which you can then look up in Appendix E. Using a library without error-message text results in an executable file that is about 2K smaller than the default.

10.3 Compiling and Linking Strategies

The following paragraphs discuss compiling and linking options that you can specify to reduce the sizes of executable files.

10.3.1 Using the Debug (/4Yb) and Integer-Size (/4I) Options

Two command-line options can significantly affect the amount of memory your source program will generate: the **/4Yb** (debug) and **/4I** (integer-size) options.

Note

The **/4Yb** and **/4I2** options correspond to the **$DEBUG** and **$STORAGE:2** metacommands, respectively, and have the same effects.

A program that includes the information generated by the debug option may contain up to 40 percent more code than the same program without the debugging information. Thus, after you have successfully compiled, linked, and run a program, you can reduce the program's size and improve its execution speed by recompiling without the **/4Yb** option (or by removing the **$DEBUG** metacommands and recompiling).

If your program does not require 4 bytes of storage for **INTEGER** and **LOGICAL** variables, you can give the **/4I2** option at compile time to reduce the size of generated code. This option allocates 2 bytes (instead of 4, the default) for each **INTEGER** and **LOGICAL** variable that you declare in the source program without an explicit length specification. This allocation cuts down on code size because less code is required to perform arithmetic on 2-byte values than on 4-byte values.

See Sections 3.3.9.2 and 3.3.10 for more information about the use of the **/4Yb** and **/4I2** options, respectively. The **$DEBUG** and **$STORAGE** metacommands are discussed in Sections 6.2.1 and 6.2.13, respectively, in the *Microsoft FORTRAN Compiler Language Reference.*

10.3.2 Using 8087/80287 Math Options

When you compile your program, you can specify a floating-point option that selects the math package that the program will use and the default library that will be used for linking. If you have an 8087 or 80287 math coprocessor, you can choose the **/FPi87** or **/FPc87** option, both of which generate instructions in your executable file for the coprocessor. If you then link with a library that supports 8087/80287 coprocessors (**LLIBFOR7.LIB** or **MLIBFOR7.LIB**), the resulting executable files will be smaller than executable files resulting from other math options because most of the software floating-point emulation is omitted. You may be able to reduce executable-file size by up to 7K by choosing one of these options.

10.3.3 Linking Version 4.0 and Version 3.3 Modules

If you are linking modules compiled with Version 4.0 and modules compiled with Version 3.3 of Microsoft FORTRAN, making sure that the compatibility library **FORTRAN.LIB** is the first library searched will reduce the size of your executable file. You can perform either of the following operations to make sure that **FORTRAN.LIB** is searched first:

- Specify the Version 3.3 modules before the Version 4.0 modules on the **LINK** command line or in response to the "Object Files" prompt.
- If you can't specify the Version 3.3 modules first, explicitly specify **FORTRAN.LIB** as the first library on the **LINK** command line or in response to the "Libraries" prompt.

10.3.4 Using Overlays

Another technique for reducing code size is to specify certain program modules as overlays at link time. Overlays are loaded into memory only when (and if) they are needed, and they share the same space in memory. In general, programs that use overlays are smaller and require less memory than programs that do not. However, programs that use overlays may run more slowly because of the additional time needed to read and reread the overlay code from disk into memory. See Section 4.7 for more information about overlays.

10.4 Coding Strategies

The following paragraphs describe ways in which you can write source programs to minimize the amount of data space the compiler needs and to minimize the amount of code the programs generate.

10.4.1 Using Consistent File-Access and Format Types

Each access type (sequential or direct) and file format (formatted, unformatted, or binary) requires that specific supporting code be incorporated in your program. If the design of your program allows, try to restrict the program to the smallest possible number of combinations of access and format. Restricting the number of file formats used has a particularly noticeable effect on code size.

10.4.2 Specifying Edit Lists

The edit lists you give for I/O statements in your program can have a significant effect on the number of run-time routines that are loaded to support your program. You save on code size by observing the guidelines given in Sections 10.4.2.1 – 10.4.2.3.

10.4.2.1 Avoiding Left Tabbing

Using the **TL** edit descriptor (or using the **T** edit descriptor to do left tabbing) increases the size of your executable files. The run-time code used to support left tabbing is included whenever the compiler detects the explicit use of these edit descriptors, or when a variable is used to contain the format string. Thus, you can reduce the size of your executable files if you avoid using left tabbing whenever possible. (See Section 4.8.1.3 of the *Microsoft FORTRAN Compiler Language Reference* for a description of the **TL** and **T** edit descriptors.)

10.4.2.2 Using Formatted or List-Directed I/O

You will create smaller executable files if you consistently use either formatted or list-directed I/O in I/O statements, and avoid mixing the two. (See Sections 4.8 and 4.9 of the *Microsoft FORTRAN Compiler Language Reference* for information on formatted and list-directed I/O specifications, respectively.)

10.4.2.3 Using Character Variables as Format Specifiers

Microsoft FORTRAN translates format strings into a more compact internal form wherever possible. However, if the format is specified as a character variable, this translation must be carried out at run time, instead of at compile time. The run-time code to do this adds significantly to the size of the executable file. As a result, you generally create smaller executable files if you use character constants or **FORMAT** statements as format specifiers. (See Sections 4.3.7.1 through 4.3.7.7 of the *Microsoft FORTRAN Compiler Language Reference* for a description of these alternatives.)

10.4.3 Using BLOCKSIZE

Using the **BLOCKSIZE** option in **OPEN** statements allows you to choose an appropriate trade-off between execution speed and memory requirements for your program.

The **BLOCKSIZE** option gives you control over the size of the internal buffer that is associated with each file your program opens. This buffer is designed to speed up execution time when your program reads or writes a large number of small items to a file. By default, the run-time system assigns a buffer size of 1024 bytes for sequential-access files. However, you can use the **BLOCKSIZE** option to allocate a buffer of a different size.

Using the **BLOCKSIZE** option gives you the following trade-off between program speed and memory requirements:

- Larger buffer sizes speed up I/O operations but increase the amount of memory allocated for these operations.
- Smaller buffer sizes result in slower I/O, but they save memory for other purposes.

Since the buffers are only allocated when the file is opened, using the **BLOCKSIZE** option will not affect the size of the program's executable file. However, if your program is near the memory limits of the machine, a smaller buffer size might allow your program to stay within those limits.

The value that you specify for the **BLOCKSIZE** option determines the actual buffer size only indirectly. This value is rounded up to a multiple of 512 for a sequential-access file for performance reasons. Multiples of 512 are used because DOS normally formats disks into 512-byte sectors and performs I/O operations to and from disk files in accordance with these sector boundaries. Thus, I/O is more efficient if the buffer size is a multiple of 512 for sequential-access files. (See Section 5.3.38 of the *Microsoft FORTRAN Compiler Language Reference* for an explanation of the rules for determining the actual buffer sizes of direct-access and terminal files.)

For example, if you specify `BLOCKSIZE=1800` when you open a sequential file, the actual buffer size will be the next multiple of 512 not less than 1800, or 2048 (4 * 512).

10.4.4 Using Integer and Real Variables

Although most FORTRAN programs use floating-point data as well as integer data, programs that use only integer data are considerably smaller than programs that use floating-point data. Thus, if you write a program that uses only integer data (for example, a program that prints out the time but does nothing else), make sure that you either explicitly declare variables with type **INTEGER**, or use variable names that begin with the appropriate letters for variables that default to **INTEGER** type. By default, these are the letters "I" through "N"; however, you can use the **IMPLICIT** statement to specify different default letters for each type. See Section 5.3.31 of the *Microsoft FORTRAN Compiler Language Reference* for more information about this statement.

10.4.5 Arrays and EQUIVALENCE Statements

The ways in which you declare arrays and use them in **EQUIVALENCE** statements can have a significant effect on the amount of data the compiler needs to compile your program. Array usage may affect whether or not the compiler runs out of memory when processing your source files. In general, the fewer assumptions the compiler must make about the types and uses of arrays, the more efficient the compilation will be.

The following paragraphs offer suggestions for using arrays in ways that minimize the amount of data the compiler uses during compilation.

Minimizing Constant Use in Array Declarations

If possible, minimize the number of different constants you use in array declarations. If you use some constants more often than others in array declarations, place the declarations that use these constants first. These actions will improve efficiency during compilation because the compiler saves copies of the constants and reuses the copies, if possible.

- **Examples**

```
INTEGER X(11), Y(10), Z(11)

INTEGER X(11), Y(11), Z(11)
```

You compile the declarations shown in the first example above more efficiently by declaring the variables X, Y, and Z with the same bounds, as shown in the second example above.

```
INTEGER X(11), Z(11)
INTEGER Y(10)
```

In the example above, if it is necessary to declare array Y with a bound of 10, you could compile more efficiently by placing the declarations of X and Z, which have the same constant bounds, first.

Minimizing Items Declared in Each Type Statement

You should declare as few items as possible in each type statement. This action reduces the complexity of each declaration and, as a result, saves the compiler data space when processing each declaration. Minimizing the number of items per type statement has an especially significant effect for array declarations.

Declaring Array Types before Dimensioning

You should explicitly declare the types of arrays before you dimension them. Also, before you use arrays in **EQUIVALENCE** statements, make sure that each array in the statement list has been completely declared, dimensioned, and (if necessary) declared in a **COMMON** statement.

■ Examples

```
INTEGER A(10,20)

INTEGER A
DIMENSION A(10,20)

INTEGER A
COMMON A(10,20)
```

The examples above show how to declare the array A to use memory most efficiently during compilation.

```
DIM X(20)
EQUIVALENCE (X(1), J)
INTEGER X
```

In the declarations shown above, the compiler requires extra data space because the type of the array X is not given before X is used in the **EQUIVALENCE** statement.

```
EQUIVALENCE (A(10,20), B)
INTEGER A, B(20,30)
COMMON /CBA/A, /CBB/B
```

In the declarations above, the compiler requires extra data space because it cannot determine the types of A and B until it has processed the second line; it does not know that B is an array until it has processed the second line; and it does not know that A and B are in a common block until it has processed the third line. In each case, the compiler must make assumptions that increase the amount of processing and the amount of data space it needs to hold intermediate processing results. The declarations above could be rewritten as follows to improve efficiency:

```
INTEGER A(10,20), B(20,30)
COMMON /CBA/A, /CBB/B
EQUIVALENCE (A(10,20), B(1,1))
```

Chapter 11
Interfaces with Assembly Language and C

11.1 Introduction 247
11.2 Assembly-Language Interface 247
11.2.1 Creating an Assembly-Language Routine: An Example 247
11.2.1.1 Producing an Assembly Listing 248
11.2.1.2 Editing the Assembly Listing 249
11.2.1.3 Optimizing Assembly Code 252
11.2.1.4 Assembling and Testing Assembly-Language Modules 254
11.2.2 The Microsoft FORTRAN Segment Model 254
11.2.2.1 Segments 254
11.2.2.2 Groups 258
11.2.2.3 Classes 258
11.2.3 FORTRAN Argument-Passing Conventions 260
11.2.4 C Argument-Passing Conventions 262
11.2.5 Entering an Assembly-Language Routine 262
11.2.6 Return-Value Conventions 263
11.2.7 Exiting a Routine 264
11.2.8 Naming Conventions 265
11.2.9 Register Considerations 265
11.3 Mixed-Language Programming 266
11.3.1 Memory Models 267

11.3.2 Choosing a Calling Convention 267
11.3.2.1 Passing Arguments by Reference or Value 268
11.3.2.2 Using Varying Numbers of Arguments 271
11.3.3 Naming Conventions 271
11.3.4 Writing Interfaces to C from FORTRAN 273
11.3.5 Calling C Procedures from FORTRAN 275
11.3.6 Writing Interfaces to FORTRAN from C 276
11.3.7 Calling FORTRAN Procedures from C 278
11.3.8 Data Types 278
11.3.8.1 Using Tables of Equivalent Data Types 278
11.3.8.2 Integers 279
11.3.8.3 Boolean and Character Types 282
11.3.8.4 Real Numbers 283
11.3.8.5 Passing Strings 284
11.3.8.6 Pointers 286
11.3.8.7 Arrays and Huge Arrays 288
11.3.8.8 Structures 290
11.3.8.9 Procedural Parameters 293
11.3.9 Return Values 293
11.3.10 Sharing Data 294
11.3.11 Input and Output 295
11.3.12 Run-Time Library Considerations 295
11.3.12.1 Accessing system and spawnlp Functions in FORTRAN Libraries 296
11.3.12.2 FORTRAN Libraries and Future Versions of Microsoft C 298
11.3.12.3 Linking Considerations 298
11.3.13 Error Messages 300

11.1 Introduction

The Microsoft FORTRAN Compiler can create object files that other languages can use, and object files created by the Microsoft FORTRAN Compiler can be linked with object files created by other languages.

This chapter first tells how to mix assembly-language modules with FORTRAN modules. This is a powerful technique for preparing assembly-language libraries for FORTRAN or for using FORTRAN routines in assembly-language programs.

The chapter also shows how to mix object files created with Microsoft FORTRAN and Microsoft C.

11.2 Assembly-Language Interface

This section explains how to use 8086/8088 assembly-language routines with FORTRAN language programs and functions. First, Section 11.2.1 gives an example of how to compile a FORTRAN routine using the **/Fa** option to create an assembly listing which is then used as the basis for an assembly-language routine. Next, Sections 11.2.2 – 11.2.9 detail the segment model used by the Microsoft FORTRAN Compiler and explain the rules for calling assembly-language routines from FORTRAN language programs and FORTRAN routines from assembly-language programs.

11.2.1 Creating an Assembly-Language Routine: An Example

You can often significantly speed up a FORTRAN function or subroutine by rewriting it in assembly language. The steps in this process are listed below:

1. Produce an assembly-language listing of the function using the **/Fa** or **/Fc** option of the compiler.
2. Edit this code to make it understandable. Insert comments and mnemonic symbols.

3. Optimize the assembly code.
4. Assemble the assembly-language module and test it by calling it from FORTRAN.

The following sections show how to do this for a function that returns the integer square root of its argument.

11.2.1.1 Producing an Assembly Listing

Assume you have written the following function to calculate integer square roots. The algorithm is slow for very large numbers, so the function is written for two-byte integers. Line numbers are shown for later reference.

```
 1:        integer*2 function isqrt (sqr)
 2:        integer*2 sqr, tester, guess
 3:        tester=1
 4:        guess=sqr
 5: 50     if (guess.ge.0) then
 6:            guess = guess-tester
 7:            tester = tester+2
 8:            goto 50
 9:        endif
10:        guess = isha(tester,-1)
11:        if ((guess**2 - guess + 1).GT.sqr) then
12:           isqrt = guess-1
13:        else
14:           isqrt = guess
15:        endif
16:        return
17:        end
```

During testing, you call this function from the following separate module called `test`:

```
1:        integer*2 i, j
2:        write(*,*) 'Enter number:'
3:        read(*,*) i
4:        j = isqrt(i)
5:        a = sqrt(i)
6:        write (*,*) 'The integer square root is ',j
7:        write (*,*) 'The real square root is ',a
8:        stop
9:        end
```

11.2.1.2 Editing the Assembly Listing

To produce an assembly listing (without linking) for the function `isqrt`, compile with the following command line:

```
FL /Fa isqrt.for /c
```

The `isqrt` function produces the following assembly-language listing:

```
;           Static Name Aliases
;
;           $S15_TESTER         EQU     TESTER
;           $S16_GUESS          EQU     GUESS
            TITLE   isqrt
;           NAME    isqrt.for

            .287
ISQRT_TEXT  SEGMENT  BYTE PUBLIC 'CODE'
ISQRT_TEXT  ENDS
_DATA       SEGMENT  WORD PUBLIC 'DATA'
_DATA       ENDS
CONST       SEGMENT  WORD PUBLIC 'CONST'
CONST       ENDS
_BSS        SEGMENT  WORD PUBLIC 'BSS'
_BSS        ENDS
DGROUP      GROUP   CONST,      _BSS,   _DATA
            ASSUME  CS: ISQRT_TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_BSS      SEGMENT
$S15_TESTER DW 01H DUP (?)
$S16_GUESS  DW 01H DUP (?)
_BSS      ENDS
ISQRT_TEXT      SEGMENT
; Line 3
            PUBLIC  ISQRT
ISQRT       PROC FAR
            push    bp
            mov     bp,sp
            sub     sp,2
            push    si
;           SQR = 6
;           ISQRT = -2
; Line 3
            mov     si,1
; Line 4
            les     bx,[bp+6]  ;SQR
            mov     cx,es:[bx]
; Line 5
            jmp     SHORT $L20003
$L20001:
; Line 6
            sub     cx,si
; Line 7
            add     si,2
; Line 8
$L20003:
            or      cx,cx
            jge     $L20001
            mov     $S16_GUESS,cx
            mov     $S15_TESTER,si
```

```
; Line 10
                mov     ax,si
                cwd
                sar     dx,1
                rcr     ax,1
                mov     $S16_GUESS,ax
; Line 11
                cwd
                mov     cx,ax
                mov     bx,dx
                imul    ax
                sub     ax,cx
                sbb     dx,bx
                add     ax,1
                adc     dx,0
                les     bx,[bp+6]   ;SQR
                mov     cx,ax
                mov     ax,es:[bx]
                mov     bx,dx
                cwd
                cmp     bx,dx
                jl      $L23__BLOCK1
                jg      $L20010
                cmp     cx,ax
                jbe     $L23__BLOCK1
$L20010:
; Line 12
                mov     ax,$S16_GUESS
                dec     ax
                jmp     SHORT $L20011
$L23__BLOCK1:
; Line 14
                mov     ax,$S16_GUESS
$L20011:
                mov     [bp-2],ax   ;ISQRT
; Line 16
                cwd
                pop     si
                mov     sp,bp
                pop     bp
                ret     4

ISQRT           ENDP
ISQRT_TEXT      ENDS
END
```

You can assemble this code as is with the Microsoft Macro Assembler, **MASM**. However, there is little point in doing so, since it will produce exactly the same object file as the FORTRAN compiler does.

The first step in optimizing assembler code is to understand it thoroughly. The following code shows the assembler listing with comments and mnemonic symbols that clarify its purpose. No changes have been made in the code itself. The segment setup is not shown in the example.

```
_BSS        SEGMENT
MEM_TESTER  DW      01H DUP (?)     ; Memory version of TESTER
MEM_GUESS   DW      01H DUP (?)     ; Memory version of GUESS
_BSS        ENDS
ISQRT_TEXT  SEGMENT
; Line 3
            PUBLIC  ISQRT
ISQRT       PROC    FAR
            push    bp
            mov     bp,sp           ; Save stack pointer
            sub     sp,2            ; Make room for local ISQR
            push    si              ; Save si
SQR         EQU     [bp+6]          ;
ISQR        EQU     [bp-2]
; Line 3
            mov     si,1            ; Initialize TESTER (in si)
; Line 4
            les     bx,SQR          ; Load address of SQR into es:bx
            mov     cx,es:[bx]      ;   and copy to GUESS in cx
; Line 5
            jmp     SHORT z_chk     ; Start at end of loop
loop:
; Line 6
            sub     cx,si           ; Subtract TESTER from GUESS
; Line 7
            add     si,2            ; Add 2 to TESTER
; Line 8
z_chk:
            or      cx,cx           ; Is guess >= 0?
            jge     loop            ; If so, do it again...
            mov     MEM_GUESS,cx    ; ...else put GUESS in memory
            mov     MEM_TESTER,si   ; Put TESTER in memory
; Line 10
            mov     ax,si           ; Copy TESTER
            cwd                     ;   to dx:ax
            sar     dx,1            ; Shift right to
            rcr     ax,1            ;   divide by 2
            mov     MEM_GUESS,ax    ; Put GUESS in memory
; Line 11
            cwd                     ; Extend GUESS in ax to dx:ax
            mov     cx,ax           ; Save a copy of GUESS
            mov     bx,dx           ;   in bx:cx
            imul    ax              ; Square GUESS
            sub     ax,cx           ; Subtract copy of GUESS in
            sbb     dx,bx           ;   bx:cx from square in dx:ax
            add     ax,1            ; Add 1 to the result
            adc     dx,0            ;   in ax:dx
            les     bx,SQR          ; Load address of SQR into es:bx
            mov     cx,ax           ; Save low byte of result
            mov     ax,es:[bx]      ; Move SQR into ax
            mov     bx,dx           ; Save high byte of result -
                                    ;   result now in bx:cx
            cwd                     ; Sign extend SQR to dx:ax
            cmp     bx,dx           ; Compare high bytes
                                    ;   of result and SQR
            jl      low             ; If lower, result is smaller
            jg      high            ; If higher, result is larger
            cmp     cx,ax           ; If equal, compare low bytes
            jbe     low             ; If lower, result is smaller
```

```
high:
; Line 12
            mov     ax,MEM_GUESS    ; Get GUESS from memory
            dec     ax              ;   and decrement
            jmp     SHORT done      ; That's all
low:
; Line 14
            mov     ax,MEM_GUESS    ; Get GUESS from memory
done:
            mov     ISQR,ax         ; Return ISQR to stack
; Line 16
            cwd                     ; Sign extend ISQR to dx:ax
            pop     si              ; Restore si
            mov     sp,bp           ; Restore stack pointer
            pop     bp
            ret     4

ISQRT       ENDP
ISQRT_TEXT  ENDS
            END
```

11.2.1.3 Optimizing Assembly Code

The FORTRAN compiler generates code that must fit any situation. The assembly-language programmer can write code that is tailored to a specific problem.

There is no standard procedure for optimizing assembly code. Every programmer develops different techniques. However, some general principles are listed below:

- Replace memory references with registers whenever possible. Try to load memory variables into registers at the start of the routine and restore them if necessary at the end. In particular, avoid using memory references for values that will change inside loops. The **SI** and **DI** registers are slightly less efficient, since they must be saved and restored if you use them.
- Examine control structures carefully. Often, you can improve on the way the compiler organizes loops and jumps.

The following example shows one way of optimizing the `isqrt` function. Notice that labels are declared public for easier debugging.

```
            TITLE   isqrt

ISQRT_TEXT  SEGMENT BYTE PUBLIC 'CODE'
ISQRT_TEXT  ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
_DATA       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
CONST       ENDS
_BSS        SEGMENT WORD PUBLIC 'BSS'
_BSS        ENDS
DGROUP      GROUP   CONST,_BSS,_DATA
            ASSUME  CS:ISQRT_TEXT,DS:DGROUP,SS:DGROUP,ES:DGROUP
ISQRT_TEXT  SEGMENT

            PUBLIC  ISQRT,loop,z_chk,low
ISQRT       PROC    FAR
            push    bp
            mov     bp,sp       ; Save stack pointer

; Register use:
;           GUESS  in CX
;           TESTER in AX
;           SQUARE in BX

; Load values into registers

            mov     ax,1        ; Initialize TESTER to 1
            les     bx,[bp+6]   ; Get address of SQUARE
                                ;   from parameter
            mov     cx,[bx]     ; Move SQUARE to GUESS
            mov     bx,cx       ;   and copy to bx
            jmp     SHORT z_chk ; Start at end of loop

; Refine guess

loop:       sub     cx,ax       ; Subtract TESTER from GUESS
            add     ax,2        ; Add 2 to TESTER
z_chk:      or      cx,cx       ; Is GUESS is >= 0?
            jge     loop        ; If so, do it again...
                                ; ...else GUESS is right
;Calculate root

            sar     ax,1        ; Divide TESTER by 2
            mov     cx,ax       ;   and copy to GUESS

; Adjust root

            imul    cx          ; Square GUESS
            sub     ax,cx       ; Subtract GUESS from TESTER
            inc     ax          ;   and add 1
            cmp     ax,bx       ; Compare to original SQUARE
            jbe     low         ; If <= we're done...
            dec     cx          ; ...else decrement GUESS

; Return GUESS (now the square root) in AX

low:        xchg    ax,cx       ; Exchange GUESS and TESTER
            mov     sp,bp
            pop     bp
            ret     4

ISQRT       ENDP
ISQRT_TEXT  ENDS
            END
```

11.2.1.4 Assembling and Testing Assembly-Language Modules

No options are necessary when assembling modules to link with compiled FORTRAN modules. For example, use the following command to assemble `isqrt`:

```
MASM /MX isqrt;
```

You can compile and link the `test` module with the following command line. The **/Zi** and **/Od** options are used to simplify debugging with the CodeView debugger:

```
FL /Zi /Od test.for isqrt
```

You might want to write a program that times the assembly version of this function against the FORTRAN version. If you compare the `isqrt` function to the `sqrt` function in the FORTRAN run-time library, you won't see a significant difference on machines with an 8087 or 80287 coprocessor, but on machines without an 8087 coprocessor, `isqrt` will be significantly faster.

11.2.2 The Microsoft FORTRAN Segment Model

This section describes the run-time segment model used for Microsoft FORTRAN programs. Memory on the 8086/8088 processor is divided into segments, each containing up to 64K. When a program is linked, the segments are organized into groups and classes. The segments, groups, and classes of Microsoft FORTRAN programs are described in Sections 11.2.2.1 – 11.2.2.3.

11.2.2.1 Segments

Figure 11.1 shows the order of primary segments of a FORTRAN program in memory, from the highest memory location to the lowest. When you look at a map file produced by linking a FORTRAN program, you may notice other segments in addition to the names listed below. These additional segments have specialized uses for Microsoft languages and should not be used by other programs.

The **/DOSSEG** option available with Microsoft **LINK** produces the ordering shown in Figure 11.1. Since this is the default ordering for FORTRAN programs, you do not need to use **/DOSSEG** with FORTRAN programs, but you may find it useful when linking assembly-language routines.

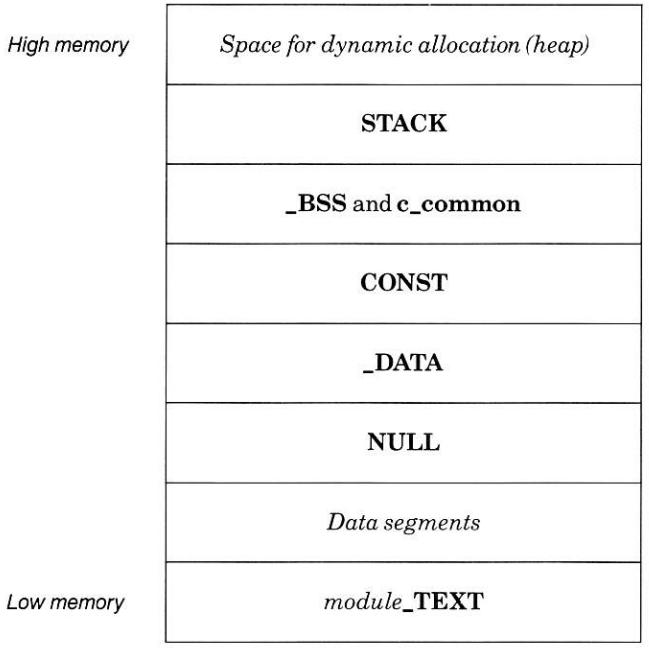


Figure 11.1 Segment Setup in FORTRAN Programs

The "heap" is the area of unallocated memory that is available for dynamic allocation by the program. Its size varies, depending on the program's other storage requirements.

Segment	Contents
STACK	The **STACK** segment contains the user's stack, which is used for saving the contexts of function and subroutine calls and for local, temporary variable storage in certain run-time support routines.
_BSS	The **_BSS** segment contains all uninitialized static data except for those that are explicitly or implicitly declared as far or huge items in the source file. In FORTRAN, static data items are local variables. (Initialized static data items are contained in the **_DATA** segment.)

c_common The **c_common** segment contains global data for medium-model programs. In FORTRAN, global data items are variables used in common blocks; these may or may not be initialized. In large-model programs, this type of data item is placed in a data segment with class **FAR_BSS**.

CONST The **CONST** segment contains all constants. These include floating-point constants, as well as segment values for data items declared far or huge in the source file or data items that are forced into their own segment by use of the **/Gt** option.

_DATA The **_DATA** segment is the default data segment where all initialized static data reside for all memory models, except data explicitly or implicitly declared far or huge, or data forced into different segments by use of the **/Gt** compiler option. (The **_BSS** segment contains uninitialized static data.)

NULL The **NULL** segment is a special-purpose segment that occurs at the beginning of **DGROUP**. (See Section 11.2.2.2, "Groups," for more information about **DGROUP**.) The **NULL** segment contains the compiler copyright notice. This segment is checked before and after program execution. If the contents of the **NULL** segment change in the course of program execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. This error may occur in FORTRAN if a program tries to store a value in an array element using an incorrect subscript, especially if the subscript is a large negative number. For example, the following program fragment causes this error:

```
C PROGRAM TO GENERATE
C 'NULL POINTER ASSIGNMENT'
      INTEGER*1 I(3)
C IADD = ADDRESS OF ARRAY I
      IADD = LOCFAR (I(1))
C ZERO OUT THE OFFSET
      IADD = IAND (IADD,#FFFF0000)
C CALCULATE ELEMENT CORRESPONDING
C TO START OF NULL SEGMENT
      IOFF = -(LOCFAR(I(1))-IADD-1)
C CLOBBER NULL SEGMENT
      I(IOFF) = 100
      END
```

This error may also arise if a C function that uses an uninitialized pointer is called. The error message

```
null pointer assignment
```

is displayed at program termination to notify the user. Although a program may appear to run correctly when this happens, it may not run under other environments.

Data segments

Initialized static far or huge data items are always placed in their own segments with class name **FAR_DATA**. Initialized, named, common data items are always placed in their own segments with class name **$***cbname*; initialized, blank, common data items are placed in their own segments with class name **$COMMQQ**. This arrangement allows the linker to combine these data items so that they all come before **DGROUP**. Uninitialized static and global far data items are placed in segments that have class **FAR_BSS**. Again, this arrangement allows the linker to place these data items between the *module*_**TEXT** segment or segments and **DGROUP**. Uninitialized huge items are placed in segments with class **HUGE_BSS**. In large- and huge-model programs, global uninitialized data are treated as though declared with the **FAR** attribute (unless specifically declared with the **NEAR** attribute) and given class **FAR_BSS**.

*module*_**TEXT**

The *module*_**TEXT** segment is the code segment. Each module is allocated its own text segment. The segments are not combined, so there are multiple text segments.

When implementing an assembly-language routine to call or be called from a FORTRAN program, you will probably refer to the **_TEXT** and **_DATA** segments most frequently. The code for the assembly-language routine should be placed in the *module*_**TEXT** segment. Data should be placed in the segment appropriate for their use, as described above. Usually this is the default data segment, **_DATA**.

11.2.2.2 Groups

All segments with the same group name must fit into a single physical segment, which is up to 64K long. This allows all segments in a group to be accessed through the same segment register. The Microsoft FORTRAN Compiler defines one group named **DGROUP**.

The **NULL**, **_DATA**, **CONST**, **_BSS**, **c_common**, and **STACK** segments are grouped together in the data group called **DGROUP**. This allows the compiler to generate code for accessing data in each of these segments without constantly loading the segment values or using many segment overrides on instructions. **DGROUP** is addressed using the **DS** or **SS** segment register.

In large- and huge-model programs, and in medium-model programs that use **FAR** data declarations with the **FAR** attribute, the **ES** register is used to allow the program to access data outside the default data segment.

The names of all text segments must end with the suffix **_TEXT**. The text segments are not grouped.

11.2.2.3 Classes

All segments with the same class name are loaded next to each other. Table 11.1 gives the alignment type, combine class, class name, and group for each segment discussed above.

Table 11.1

Segments, Groups, and Classes for Standard Memory Models

Memory Model	Segment Name	Align-ment Type	Combine Class	Class Name	Group
Medium	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data segments[1]	para	private	FAR_DATA	
	Data segments[2]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Large	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data segments[3]	para	common	$*cbname*	
	Data segments[4]	para	common	$COMMQQ	
	Data segments[5]	para	private	FAR_DATA	
	Data segments[6]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

Table 11.1 *(continued)*

Memory Model	Segment Name	Align-ment Type	Combine Class	Class Name	Group
Huge	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data segments[3]	para	common	$*cbname*	
	Data segments[4]	para	common	$COMMQQ	
	Data Segments[5]	para	private	FAR_DATA	
	Data Segments[6]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

[1] Segment(s) for initialized far or huge data

[2] Segment(s) for uninitialized far or huge data

[3] Segment(s) for initialized, named common data (segment names *cbname*$A, *cbname*$B...; group $*cbname*)

[4] Segment(s) for initialized, blank common data

[5] Segment(s) for initialized static data

[6] Segment(s) for uninitialized global and static data

11.2.3 FORTRAN Argument-Passing Conventions

To receive values from or pass values to a FORTRAN routine, an assembly-language routine must follow the FORTRAN argument-passing conventions. By default, FORTRAN programs pass arguments by reference, although passing by value can be specified.

If a FORTRAN program passes arguments by reference to an assembly-language routine, it pushes the addresses of the actual arguments onto the stack from left to right; that is, the address of the first argument is pushed first. If the argument is passed by value, the value itself is pushed onto the stack.

These addresses occupy either one or two words on the stack, depending on the memory model and whether the argument was declared with a **NEAR**, **FAR**, or **HUGE** attribute. For far argument addresses, the offset is pushed first, then the segment value.

For routines that return values other than integer or logical values, a "hidden" last argument is pushed. This argument is the address of the temporary variable to which the result of the routine will be returned.

Routines that have formal arguments of type **CHARACTER*(*)** use their own mechanism for communicating the length of the actual arguments. When the compiler compiles the calling routine, it builds an array of 16-bit integers on the stack. Elements of this array correspond in number and order to the actual **CHARACTER*** *n* arguments in the call. Each element contains the length, in bytes, of the corresponding argument. The address of the array is assigned to a global variable named **__FIclenv**. (The called routine usually copies this address as part of the entry sequence, since it may be reassigned as a result of another call with **CHARACTER*** *n* arguments before it can be used to obtain length.) If the length of the formal argument is known, the length given by the corresponding element in the array is ignored.

Note

This mechanism means that routines compiled with Version 3.3 of the Microsoft FORTRAN Compiler can call routines compiled with Version 4.0, provided that all formal arguments are declared with fixed lengths so that the Version 4.0 code does not contain references to **__FIclenv**. However, Version 3.3 routines cannot call Version 4.0 routines if the Version 4.0 routines use **CHARACTER*(*)** formal arguments. This is because the **__FIclenv** mechanism does not exist in Version 3.3, and the results of the call are undefined.

Before an assembly-language routine returns control to a FORTRAN routine, it is responsible for removing arguments from the stack.

11.2.4 C Argument-Passing Conventions

The C argument-passing conventions, enabled for individual subprograms by use of the **C** attribute, cause FORTRAN programs to make calls in which function and subroutine arguments are pushed onto the stack right to left; that is, the last argument is pushed first. When this alternative calling sequence is enabled, the calling routine is responsible for removing the arguments from the stack. This convention makes it possible to use functions and subroutines with variable-length argument lists.

11.2.5 Entering an Assembly-Language Routine

Assembly-language routines that receive control from FORTRAN function calls should preserve the contents of the **BP**, **SI**, and **DI** registers and set the **BP** register to the current **SP**-register value before proceeding with their tasks. The routine is required to save **SI** and **DI** only if it changes them.

If the assembly-language routine changes the contents of the **SS**, **DS**, or **CS** registers, the register values should be saved on entry and restored on exit. In FORTRAN programs, the values of **SS** and **DS** are always equal.

The following example illustrates the recommended instruction sequence for entry to an assembly-language routine:

```
ENTRY:    push bp     ;save caller's frame pointer (BP)
          mov  bp,sp  ;frame pointer points to old BP
          sub  sp,8   ;allocate local variable space on stack
          push si     ;required only if routine changes si
          push di     ;required only if routine changes di
        .
        .
        .
```

This is the same sequence used by the Microsoft FORTRAN Compiler; in fact, you can generate an assembly-language listing such as the above by compiling your FORTRAN program with the **/Fa** or **/Fc** options. See Section 3.3.7, "Creating Listing Files," for more information.

Note that the **push** instructions in this sequence are only necessary if the assembly-language routine changes the contents of the **SI** and **DI** registers. The compiler uses these registers to store heavily used values for fast access during optimization.

For each type of function call, Table 11.2 shows where the address of the first argument appears on the stack when the FORTRAN calling convention is used.

Table 11.2

First Argument Address on Stack for FORTRAN Calling Convention

Number of Arguments	Call Type	First Argument Address Location
1	Near	[**BP**+4]
1	Far	[**BP**+6]
(2-byte)	Near	[**BP**+6]
(2-byte)	Far	[**BP**+8]
(4-byte)	Near	[**BP**+8]
(4-byte)	Far	[**BP**+10]

11.2.6 Return-Value Conventions

Assembly-language routines that return values to a FORTRAN program or receive return values must follow the FORTRAN return-value conventions. These conventions specify the registers where FORTRAN programs expect to find return values of each data type. In some cases, the return-value conventions depend on whether the FORTRAN or C calling conventions are used to call the assembly-language routine.

Table 11.3 shows the FORTRAN return-value conventions.

Table 11.3

FORTRAN Return-Value Conventions

Return-Value Type	Calling Convention	Register(s) Used
INTEGER*1, **LOGICAL*1**	FORTRAN, C	**AL**
INTEGER*2, **LOGICAL*2**	FORTRAN, C	**AX**
INTEGER*4, **LOGICAL*4**	FORTRAN, C	High-order word in **DX**; low-order word in **AX**
REAL*4, **REAL*8**, **COMPLEX*8**, **COMPLEX*16** **CHARACTER*n**†	FORTRAN	Temporary variable created by the calling program. A near offset to this variable is stored at **BP**+4 on the stack for medium-model programs or at **BP**+6 for programs that use other memory models. This address is returned in **AX**. For large and huge models, the value in **SS** is returned in **DX** to provide a far pointer.
	C	Temporary variable allocated in **DGROUP**. This address is returned in **AX**. For large and huge models, the value in **DS** is returned in **DX** to provide a far pointer.

† The length specified in the calling function is placed in the length descriptor block addressed by **__FIclenv**. (See Section 11.2.3, "FORTRAN Argument-Passing Conventions," for more information about **__FIclenv**.)

11.2.7 Exiting a Routine

Assembly-language routines that return control to FORTRAN programs should restore the values of the **BP**, **SI**, and **DI** registers before returning control. (The contents of the **SI** and **DI** registers must be restored only if the entry sequence pushed them.) The following example illustrates the recommended instruction sequence for exiting a routine called by a large-model program:

```
EXIT:     pop si       ;required only if si saved on entry
          pop di       ;required only if di saved on entry
          mov  sp,bp   ;remove local variable space
          pop  bp      ;restore caller's frame pointer
          ret num ;appropriate to type of call
```

The sequence uses the `ret` *num* instruction (return and pop *num* bytes off the stack, where *num* is the size in bytes of all arguments) to clear the arguments from the stack. It does not change the **AX**, **BX**, **CX**, or **DX** registers or any of the segment registers.

Note

If the external declaration of the assembly-language routine contains the **C** attribute, then the FORTRAN routine that called the assembler routine must remove the arguments from the stack after the assembler routine returns control to the FORTRAN routine. In this case, the `ret` *num* instruction at the end of the preceding example must be replaced with the `ret` instruction (simple return).

11.2.8 Naming Conventions

To avoid conflicts with internal names used by the FORTRAN compiler, you must observe certain naming conventions in assembly-language routines.

The Microsoft FORTRAN Compiler reserves some names beginning with two underscores (__) and six-letter names ending with "QQ" for internal use. Avoid using names with two leading underscores or two trailing Qs in your assembly-language routines, as these names may conflict with internal names.

11.2.9 Register Considerations

Assembly-language routines that are called by FORTRAN programs must obey certain rules regarding the use of registers.

If you use the **/Os** or **/Op** option with the **FL** command to optimize your program, the compiler uses the **SI** and **DI** registers to store frequently accessed variables during optimization. An assembly-language routine that changes the **SI** and **DI** registers is responsible for saving their contents on entry and restoring them before exiting.

The Microsoft FORTRAN Compiler assumes that the direction flag is always cleared. If your assembly-language routine sets the direction flag, be sure to clear it, using the **CLD** instruction, before returning.

If the assembly-language routine changes the contents of the **SS**, **DS**, and **CS** registers, their values should be saved on entry and restored at exit. The values of **SS** and **DS** are always equal, except if **DS** is temporarily changed to a different value to allow the program to access data outside the default data segment. The **ES** register may also be used in these cases.

11.3 Mixed-Language Programming

Microsoft FORTRAN (Version 3.3 or later) and Microsoft C (Version 4.0 or later) allow programs written in either of these languages to call routines written in the other language.

Mixed-language programming offers the following advantages:

- You can use libraries of procedures written in different languages.

  For example, you can access the standard Microsoft C run-time libraries from programs written in FORTRAN. Also, you can access many proprietary libraries available for use with Microsoft FORTRAN from programs written in Microsoft C.

  To use a library written for a particular language, you must have the standard run-time library supplied with that language's compiler for the appropriate memory model. For example, to use a proprietary FORTRAN library from C, you need the appropriate *x***LIBFOR***x***.LIB** library supplied with the Microsoft FORTRAN Compiler, as well as the proprietary library itself.

- You can use features not available in your language. For example, some interfaces, such as those that use C structures, are not compatible with FORTRAN.

- If you write your own libraries, you can now produce one library that is compatible with both languages.

To ensure compatibility, you must pay close attention to the guidelines given in this section.

11.3.1 Memory Models

Version 4.0 of Microsoft FORTRAN and Version 4.0 of Microsoft C both support the medium, large, and huge memory models. Version 3.3 of Microsoft FORTRAN supports only the large memory model. In general, if a program written in one of these languages calls a routine written in the other language, both programs should use the same memory model.

Version 4.0 of FORTRAN does not support the small and compact memory models that are supported in Version 4.0 of C, and FORTRAN programs cannot call small- or compact-model C functions by default.

11.3.2 Choosing a Calling Convention

The default conventions for passing arguments in Microsoft FORTRAN and Microsoft C are different in the following respects:

- The order in which arguments are pushed onto the stack. FORTRAN pushes arguments onto the stack in the order in which they appear in the procedure declaration. C pushes its arguments in the reverse order.
- The code that restores the stack when a procedure returns. Under the FORTRAN convention, the called procedure must restore the stack; under the C convention, the calling procedure must restore the stack. The FORTRAN convention is slightly faster and produces less code. The C convention allows you to use a varying number of arguments. (Because the first argument is always the last one pushed, it is always nearest the top of the stack, and it always has the same address relative to the start of the frame.) These conventions are incompatible.
- Which arguments are passed by reference and by value. See Section 11.3.2.1, "Passing Arguments by Reference or Value," for a discussion of these differences.

If you control both the calling and the called code, you can choose which calling convention to use. If you intend to pass varying numbers of arguments, you must use the C calling convention. See Section 11.3.2.2, "Using Varying Numbers of Arguments," for more information. Otherwise, you may want to use the convention of the language that you use most often, so that you can usually use the default calling convention.

To make calls from one language to another, you must tell the compiler which convention to use. Microsoft FORTRAN and C provide ways to specify which convention you use, both when you call an external procedure and when you define a public procedure. Table 11.4 indicates how to specify calling conventions from each language.

Table 11.4

Specifying Calling Conventions

Calling Convention	Language Calling from	Attributes or Keywords to Use
C	FORTRAN	**C** attribute on **INTERFACE** statement
	C	Default or **cdecl** keyword with **/Gc** option
FORTRAN	FORTRAN	Default
	C	**fortran** keyword on procedure declaration, or **/Gc** option

- **Examples**

```
INTERFACE TO INTEGER*4 FUNCTION FOO[C] (BAR)
```

The FORTRAN **INTERFACE** statement above specifies that the C function `foo` will be called using the C conventions.

```
extern int fortran foradd(int *,int *);
```

The C **extern** statement above specifies that the FORTRAN function FORADD will be called using the FORTRAN conventions.

11.3.2.1 Passing Arguments by Reference or Value

Passing an argument by reference means that the argument's address is passed rather than the argument itself. Procedures access the argument's value through the address, so that any changes a procedure makes to the argument affect the stored value.

When an argument is passed by value, a copy of the argument is placed on the stack when the procedure is called. The procedure can change the value of the argument without affecting the original value from which the copy was taken.

You must decide whether to pass each argument by value or by reference. If you pass by reference, you also must choose whether to pass a far address (segment and offset) or a near address (offset only).

If the called procedure needs to change the actual argument's value as a way of returning a result, you must pass that argument by reference. Passing an argument by value protects the argument from accidental updating and, for arguments smaller than 4 bytes, can be more efficient.

The following list describes the default argument-passing conventions for FORTRAN and C:

- By default, FORTRAN passes all arguments by reference (including constants and expressions).

  If you declare a FORTRAN procedure with the **C** attribute, the default changes: all arguments to that procedure are passed by value.

- By default, C passes arrays by reference and all other arguments by value. C can also pass pointers as arguments to a procedure. The procedure can use the pointers to change stored values, producing the same effect as passing by reference.

Table 11.5 shows how to declare arguments in FORTRAN and C to override any of these defaults.

Table 11.5

Overriding Default Argument-Passing Conventions

Argument	C	FORTRAN
Far address	**far** pointer	**REFERENCE** and **FAR** or **HUGE** attributes
Short address	**near** pointer	**REFERENCE**, **NEAR** attributes
Value	**struct** including the array	**VALUE** attribute

■ Example

Assume that you are using the C calling conventions. Table 11.4 shows which attributes and keywords are necessary to use the C calling conventions.

- When calling from FORTRAN, specify the **C** attribute on the **INTERFACE** statement.
- When calling from C, the C calling conventions are the default, unless your program has been compiled with the **/Gc** option, or the function your program is calling has been declared with the **fortran** keyword. (See Section 11.3.2, "Choosing a Calling Convention.")

Assume that you want to pass an integer argument, `x`, using a far address. Compatibility of data types is discussed in Section 11.3.8; for now, assume that the C **int** type and the FORTRAN **INTEGER** type are equivalent. Table 11.5 shows that when declaring the argument `x` in your C procedure, you should use a **far** pointer of the appropriate type (in this case, **int**) since the large memory model is the default for FORTRAN.

The following is the C declaration:

```
int *x;
```

For the FORTRAN procedure, specify the **REFERENCE** attribute, as shown below, since you are using the C calling conventions but still want to pass `x` by reference:

```
      INTEGER X[REFERENCE]
```

If you want to pass using a short address instead, the appropriate declaration in C is as follows:

```
int near *x;
```

and in FORTRAN it is the following:

```
      INTEGER X[REFERENCE,NEAR]
```

You follow the same steps when declaring arguments even if you are using other calling conventions. If you are passing arguments using FORTRAN calling conventions, use the constructs described in Table 11.5 when declaring arguments.

11.3.2.2 Using Varying Numbers of Arguments

If you are going to use varying numbers of arguments in a FORTRAN routine, keep these factors in mind:

- If the called routine is written in FORTRAN, the number of actual arguments must be less than or equal to the number of formal arguments.

  It is difficult in FORTRAN to access arguments that have not been formally defined. You can use the **VARYING** attribute to allow fewer arguments to be passed than the number defined. (In C, arguments that have not been formally defined can be accessed through pointer arithmetic and dereferencing.)

- If the subprogram declaration appears before the call to the subprogram, you can specify the **C** and **VARYING** attributes in the subprogram declaration. Otherwise, you must use the **C** and **VARYING** attributes in the FORTRAN **INTERFACE** statement, which must precede the use of the subprogram.

  The **VARYING** attribute tells the Microsoft FORTRAN Compiler not to verify that the number of actual arguments and the number of formal arguments are the same. If an actual argument is specified for a formal argument, the compiler still checks for type compatibility between the arguments according to the usual rules of the calling procedure's language.

■ **Example**

```
INTERFACE TO SUBROUTINE SET1S[C,VARYING] (A, B, C, D)
```

The **INTERFACE** statement above specifies a varying number of arguments to the subroutine SET1S.

11.3.3 Naming Conventions

Because of the differences in default naming conventions between Microsoft FORTRAN and Microsoft C, you should use the following rules to make sure that the compilers handle all the necessary adjustments in names:

- If you are using any FORTRAN routines, either specify the **/4Nt** option when you compile, include the **$NOTRUNCATE** metacommand in your source file, or make sure that all names are six characters or less in length.

- Avoid using uppercase characters in C identifiers. If you must use uppercase characters, do *not* specify the **/NOIGNORECASE** (**/NOI**) option to the **LINK** command, and do not use other identifiers that have the same spelling as the uppercase or mixed-case C identifiers. (For example, if one C identifier is `AnExample`, don't use `anexample`, `ANEXAMPLE`, or `AnExAmPlE` as an identifier.)

If you cannot follow these rules, you must make certain adjustments yourself. The remainder of this section explains the default naming conventions for FORTRAN and C and how certain attributes and keywords affect those naming conventions. This information will help you to solve any special problems in naming.

In both FORTRAN and C, names appear differently in the object and source files. The following elements of their respective naming conventions are different:

Element	Differences
Case	In FORTRAN, any lowercase letters in a public identifier are changed to uppercase before the name is inserted in the object file. By default, C names are not transformed in this way; however, at link time you can specify that case distinctions should be ignored.
Length	In FORTRAN, by default, names are truncated to six significant characters.
Underscores	In C, public names are always prefixed with an underscore character (_) before they are inserted in the object file.

These differences in naming conventions mean that default FORTRAN public names do not correspond to default C public names. Certain attributes and keywords can help you make names correspond.

If you want FORTRAN names to follow the C conventions, specify the **C** attribute with the names of FORTRAN procedures, interfaces, or named common blocks. When you specify this attribute, uppercase letters in each name are changed to lowercase and a leading underscore is added. However, the name is still truncated to six characters. To use a longer name, specify the **/4Nt FL** option when you compile (or include the **$NOTRUNCATE** metacommand in your source file). In this case, names, including alias names, cannot exceed 31 characters. To specify external C routines that have uppercase letters in their identifiers, use the **ALIAS** attribute.

If you use the **fortran** keyword in C, the name is changed to uppercase and the leading underscore is not added to the name. All such names must have unique spellings.

Note that, in FORTRAN, if an **INTERFACE** statement and the subprogram referred to in that **INTERFACE** statement are in the same module, the same types must be used for the arguments in each. In addition, the **INTERFACE** statement must appear before the subprogram. An error message is generated if you violate either of these rules.

11.3.4 Writing Interfaces to C from FORTRAN

The FORTRAN **INTERFACE** statement allows you to declare external procedures in C from FORTRAN using the following procedure:

1. Find the declaration of the C procedure.
2. Build an **INTERFACE** program unit as described below:
   a. Determine the attributes and type for the procedure, and decide which calling convention to use.
   b. Determine the attributes and types for the arguments.
3. Add the **INTERFACE** statement to the program.

The final step, calling the C procedure, is described in Section 11.3.5.

■ Example

Suppose that you want to access the following C procedure named `time` from a FORTRAN program:

```
long time(long *);
long time(tloc);
long *tloc;
{
   printf(" t = %ld\n",*tloc);
   return ((*tloc)*2);
}
```

In this example, the declaration of the C procedure `time` looks like this:

```
long time (tloc);
long *tloc;
```

Step 1 is to decide which attributes and types to use for the procedure and arguments. You must first determine which FORTRAN type is equivalent to the type of the procedure `time`. The first word in the C procedure declaration `long time (tloc);`, shows that `time` has type **long**. Referring to Table 11.10 in Section 11.3.8.2, "Integers," you can see that the FORTRAN **INTEGER*4** type is equivalent to the C **long** type.

This gives enough information to begin Step 2 by writing the following **INTERFACE** statement:

```
INTERFACE TO INTEGER*4 FUNCTION TIME
```

Since you have no control over the C procedure, you must use the calling conventions that the procedure uses. To specify the C calling conventions, specify the C attribute as shown below:

```
INTERFACE TO INTEGER*4 FUNCTION TIME[C]
```

Now, determine which attributes and data types to use for the arguments. Since the C procedure has only one argument, `tloc`, you can write

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC)
```

Note, however, that in line 2 of the C procedure declaration, `tloc` is preceded by an asterisk, indicating that a pointer to `tloc` is being passed. In FORTRAN, you can pass a pointer to an argument by using the **LOCFAR** or **LOC** intrinsic functions, or you can pass the argument itself by reference. For now, assume that you want to pass `tloc` by reference. Because the `time` procedure was declared with the C attribute, the default is to pass `tloc` by value. To pass `tloc` by reference, add the **REFERENCE** attribute, as shown below:

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC[REFERENCE])
```

The first word in line 2 of the C procedure declaration, `long`, indicates the type of the argument `tloc`. Since the FORTRAN **INTEGER*4** type is equivalent to the C **long** type, you can finish the **INTERFACE** statement as shown below:

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC[REFERENCE])
 INTEGER*4 TLOC
 END
```

If you decide to pass a pointer to `tloc` instead of passing it by reference, follow the same procedure up to this point:

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC)
```

Since pointers are passed by value, do not specify the **REFERENCE** attribute. Pointers are normally 4-byte segmented addresses (except in medium-model programs), and the result of **LOC** is a 4-byte integer. Therefore, you must declare `tloc` to be a 4-byte integer, as shown below:

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC)
 INTEGER*4 TLOC
 END
```

Step 3, adding the **INTERFACE** unit to your program, is identical for both cases. The only rule to follow is that the **INTERFACE** unit must appear before any references to the procedure. It is usually easiest to put all **INTERFACE** statements at the beginning of the source file.

The final step, calling the procedure, is different for the case involving the **REFERENCE** attribute than for the case that uses a pointer, as described in Section 11.3.5.

11.3.5 Calling C Procedures from FORTRAN

Once you have included an **INTERFACE** statement declaring a C procedure in your FORTRAN program, you can call the C procedure in the same way as you would call a FORTRAN procedure.

- **Example**

For the example discussed in Section 11.3.4, start writing the calling routine as shown below:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
```

Remember to declare the procedure, as in the following line; otherwise, the TIME procedure would be real by default because the first letter of the procedure name implicitly types it as a real procedure:

```
INTEGER*4 TIME
```

Now, if you passed tloc by reference, you can complete the call as follows:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE (*,*) TIME (TLOC)
END
```

If you passed a pointer to tloc, your procedure call looks like this:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE(*,*) TIME(LOC(TLOC))
END
```

You could substitute the **LOCFAR** intrinsic function for the **LOC** intrinsic function. In this implementation (large model), they are identical.

Note that if time were a subroutine instead of a function, you could call that subroutine with the FORTRAN **CALL** statement.

11.3.6 Writing Interfaces to FORTRAN from C

In a C program, the **fortran** keyword can be used to declare selected procedures written in, or compatible with, FORTRAN. This keyword implies changes in the default external-naming, calling, and return-variable conventions for C.

If you want all procedures in your C program to be compatible with FORTRAN, specify the **/Gc** option with the **CL** command when you compile the program.

In C programs, you declare FORTRAN procedures in the same way as you declare C procedures: by specifying the procedure identifier, the return type, and the type and number of arguments to the procedure. (See the *Microsoft C Compiler Language Reference* for a complete discussion of the syntax of procedure declarations.)

The following additional rules apply when you use the **fortran** keyword:

- Whenever the **fortran** keyword is used in a declaration, the types of the arguments to the procedure must be declared with an argument-type list.
- The **fortran** keyword changes the item immediately to the right in a declaration.
- The special **near** and **far** keywords can be used with the **fortran** keyword in declarations. The sequences **far fortran** and **fortran far** are equivalent.

Complex declarators are allowed in **fortran** declarations, just as in C procedure declarations.

■ Examples

The following examples illustrate the syntax of **fortran** declarations. Note that these examples assume that the arguments are passed by value, the default in C (except for array arguments).

```
short fortran thing1(short, short);
```

The example above declares `thing1` to be a FORTRAN routine that takes two **short** arguments and returns a **short** value.

```
long (fortran *thing2)(void);
```

In the example above, `thing2` is declared as a pointer to a FORTRAN routine that takes no arguments and returns a **long** value. Note that **void** is used to indicate that there are no arguments.

```
short near fortran thing3(short);

short fortran near thing4(short);
```

The examples above are equivalent. The first example declares `thing3`, and the second example declares `thing4` to be a **near** FORTRAN procedure. The procedures take one **short** argument and return a **short** value.

11.3.7 Calling FORTRAN Procedures from C

To call a FORTRAN procedure from C, you must declare that procedure to be external, as shown in the following example:

```
extern void fortran m(long);
```

Note that **void** is used to indicate that there is no return value.

Once you have declared a procedure, you can call it in your program just as if the procedure were in C. Specifying the **fortran** keyword does not change the way arguments are passed by default: array arguments are still passed by reference, and nonarray arguments are still passed by value.

11.3.8 Data Types

FORTRAN and C have a variety of data types. Some are completely compatible; others require manipulation to work between languages. Sections 11.3.8.2 – 11.3.8.9 explain how specific data types differ in each language. Tables 11.6 – 11.24 show sample variable declarations that illustrate equivalent data types for each language. (Where any valid data type for the language can be used, *type* is shown as the data type.)

11.3.8.1 Using Tables of Equivalent Data Types

When you use Tables 11.6 – 11.24 to determine compatible data types for passing arguments, you must also refer back to Table 11.4, "Specifying Calling Conventions," and Table 11.5, "Overriding Default Argument-Passing Conventions."

For example, suppose that you want to pass an **INTEGER*2** argument from FORTRAN to C. Use the following procedure:

1. Choose a calling convention, as explained in Section 11.3.2. Assume that you want to use the C calling conventions. Refer to Table 11.4, "Specifying Calling Conventions," in Section 11.3.2.
2. Decide whether to pass the argument by reference or by value. Assume that you want to pass the argument by reference, using a short address. Table 11.5, "Overriding Default Argument-Passing Conventions," in Section 11.3.2.1, "Passing Arguments by Reference or Value," shows that you would use the **REFERENCE** and **NEAR** attributes in FORTRAN and a **near** pointer of the appropriate type in C.

3. Determine which data type in C is equivalent to the **INTEGER*2** type in FORTRAN. Find the table that lists signed 2-byte integers (Table 11.8). Note that **INTEGER*2** is listed as an appropriate FORTRAN data type. Check the "Notes" column to see if there is anything to watch out for when using **INTEGER*2**.
4. Look at the "C" row. You can choose between **short** and **int**, but the "Notes" column shows that **int** is machine dependent. For maximum portability, choose the C **short** type.
5. Apply the appropriate attributes and keywords to the data types, as shown below:

```
INTEGER*2 X [REFERENCE,NEAR]
```

In a FORTRAN **INTERFACE** statement declared with the **C** attribute, the statement above is equivalent to a C argument declared as shown below:

```
short near * x
```

Note that passing an argument by reference in FORTRAN corresponds to using a pointer type in C.

11.3.8.2 Integers

In C, any integer arguments shorter than **int** (such as **char**) are converted to **int** type before they are passed by value. Unsigned integer types shorter than an **unsigned int** (such as **unsigned char**) are converted to **unsigned int** type.

To ensure that your FORTRAN routines handle C arguments correctly, you have two options:

1. Allow for the C conversions when you declare arguments to the FORTRAN procedure. This means, for example, that you must declare all integer arguments to have sizes corresponding to a C **int** or **long int**, for integer arguments larger than an **int**.
2. You can pass pointers to the arguments instead of the values themselves; that is, you can pass the arguments by reference. In the FORTRAN routine, declare the passed arguments as pointers to or reference arguments of the appropriate types, then use the pointers to access the values indirectly.

Also, note that the C **int** type is machine specific. For the 8086 family of microprocessors, the C **int** type is equivalent to the FORTRAN types **INTEGER*2** and **INTEGER[C]**.

For any given processor and operating system, variables defined with the **INTEGER[C]** type are equivalent to variables of the C **int** type as defined by the Microsoft C Compiler for the same system. This type is therefore more portable than the **INTEGER*2** type.

Tables 11.6 – 11.10 show integer data types and their equivalents in C and FORTRAN.

Table 11.6

Signed 1-Byte Integers

Language	Data Type	Notes
FORTRAN	**INTEGER*1** *X*	---
C	**char** **x*;	When passed by reference
	struct { **char** *x*;} *x*;	When passed by value

Table 11.7

Unsigned 1-Byte Integers

Language	Data Type	Notes
FORTRAN	**INTEGER*1** *X*	No "unsigned" types in FORTRAN
C	**unsigned char** **x*;	When passed by reference
	struct { **unsigned char** *x*;} *x*;	When passed by value

Table 11.8
Signed 2-Byte Integers

Language	Data Type	Notes
FORTRAN	**INTEGER*2** *X*	---
	INTEGER[C] *X*	---
	INTEGER *X*	If **/4I2** or **$STORAGE:2** is in effect
C	**short** *x*;	Machine dependent
	int *x*;	Machine dependent

Table 11.9
Unsigned 2-Byte Integers

Language	Data Type	Notes
FORTRAN	**INTEGER*2** *X*	FORTRAN has no unsigned types, so you must use **INTEGER*2**. Do not pass negative values or values greater than 32,767. Note that many unsigned operations can be performed safely on **INTEGER*2** values.
C	**unsigned short** *x*;	Machine dependent
	unsigned int *x*;	Machine dependent

Table 11.10

Signed 4-Byte Integers

Language	Data Type	Notes
FORTRAN	**INTEGER*4** *X*	---
	INTEGER *X*	If **/4I4** or **$STORAGE:4** (the default) is in effect
C	**long** *x*;	---

C also has unsigned 4-byte integers. FORTRAN does not. However, many unsigned arithmetic operations can be performed on signed variables and still yield correct results. This level of type equivalence may be sufficient for some applications.

11.3.8.3 Boolean and Character Types

Tables 11.11 and 11.12 show how Boolean and character types, respectively, are represented in C and FORTRAN.

Table 11.11

Boolean Types

Language	Data Type	Notes
FORTRAN	**CHARACTER*1** *X*	Use as for unsigned 1-byte integers: 0 = false and 1 = true. FORTRAN **LOGICAL** types are *not* equivalent. See Tables 11.22 – 11.24 for information on FORTRAN **LOGICAL** types.
C	**unsigned char** *x*;	---

Table 11.12

Character Types

Language	Data Type
FORTRAN	**CHARACTER** *X*
C	**unsigned char** *x*;

11.3.8.4 Real Numbers

C passes all real arguments as double-precision values. To ensure that your FORTRAN routines handle C arguments correctly, you have the following three options:

1. You can allow for the C conversions when you declare arguments to the FORTRAN procedure. This means that you must declare all floating-point arguments as double-precision arguments (**REAL*8** in FORTRAN), and specify the **VALUE** attribute in FORTRAN.
2. You can pass pointers to the arguments instead of the arguments themselves. In the FORTRAN routine, declare the passed arguments as references to the appropriate types.
3. To avoid expansion of a **float** value to a **double** value, you can pass the value as a structure. The members of structures do not undergo type conversion when the structure is passed as an argument. For example, the following declaration defines a structure variable, `arg`, with a single **float** member:

   ```
   struct fptype {float a;} arg;
   ```

   After you declare `arg`, you can pass it as an argument. Passing such a structure as an argument in C is equivalent to passing a **REAL*4** value in FORTRAN (except that FORTRAN normally passes by reference).

Floating-point values returned to C from FORTRAN are handled as structured values.

Tables 11.13 and 11.14 show equivalent real types in C and FORTRAN.

Table 11.13
Single-Precision Real Numbers

Language	Data Type	Notes
FORTRAN	**REAL** *X*	---
	REAL*4 *X*	---
C	**float** *x*;	---
	struct { **float** *x*;} *x*;	When passed by value

Table 11.14
Double-Precision Real Numbers

Language	Data Types
FORTRAN	**REAL*8** *X* **DOUBLE PRECISION** *X*
C	**double** *x*;

11.3.8.5 Passing Strings

FORTRAN and C store string constants in memory in different ways. In order to pass strings from one language to another, you must give the compiler the appropriate information about how the string is set up.

C strings are considered arrays of characters. The null (zero-value) character marks the end of the string and is the last character of the array. For example, the string

```
String of text
```

is indicated in C as

```
unsigned char str[ ]="String of text";
```

This string is stored in memory as a 15-byte array: 14 bytes of significant text (that is, the string itself) and 1 null character that marks the end of the string, as shown in Figure 11.2.

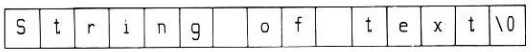


Figure 11.2 C String Stored in Memory

FORTRAN strings do not have delimiters in memory. The length of the string is determined in advance. The above string is written in FORTRAN as

```
CHARACTER*15 STR
STR='String of text'
```

It is stored in memory as 14 bytes of text, as shown in Figure 11.3.

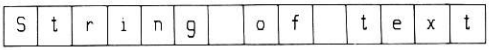


Figure 11.3 FORTRAN String Stored in Memory

Table 11.15 summarizes how each language handles string and array types. The placeholder *n* in the table is a constant, and each type occupies *n* bytes.

Table 11.15
String and Array Types

Language	Type
FORTRAN	**CHARACTER*** *n C*
	CHARACTER*1 *C(n)*
C	**unsigned char** *c*[*n*];
	struct *cstr* {**unsigned char** *c*[*n*];}*c*;

The following sections explain how to pass strings from one language to another.

Passing FORTRAN Strings to C

To pass FORTRAN strings to C, use the C string feature. When a standard FORTRAN string constant is followed by the character **C**, that string is then interpreted as a C string constant. A null character is automatically appended to the end of the string, and backslashes (\) are treated as escape characters. See Section 2.4.6.1 of the *Microsoft FORTRAN Compiler Language Reference* for information about the C string feature.

Note

In Microsoft FORTRAN, the length of a string is passed with the string. See Section 11.2.3, "FORTRAN Argument-Passing Conventions," for a description of this process.

Passing C Strings to FORTRAN

To FORTRAN, C strings are just arrays. When passing C strings to FORTRAN, allow room for the null byte at the end of the string.

11.3.8.6 Pointers

Tables 11.16 – 11.18 show equivalent pointer types for each language.

Table 11.16

Near Pointers

Language	Data Type	Notes
FORTRAN	*type OBJECT* **INTEGER*2** *X* *X*=**LOC**(*OBJECT*)	Medium model
	type OBJECT **INTEGER*2** *X* *X*=**LOCNEAR**(*OBJECT*)	Other memory models
C	*type* **near** **x*;	---

Table 11.17

Far Pointers

Language	Data Type	Notes
FORTRAN	*type OBJECT* **INTEGER*4** *X* *X*=**LOCFAR**(*OBJECT*)	Medium model
	type OBJECT **INTEGER*4** *X* *X*=**LOC**(*OBJECT*)	Other memory models
C	*type* ******x*;	---
	type **far** ******x*;	---

Table 11.18

Procedure Pointers

Language	Data Type	Notes
FORTRAN	**INTEGER*4** *PROC* **EXTERNAL** *PROC* **INTEGER*4** *X* *X*=**LOC**(*PROC*)	**EXTERNAL** must be used if you are using **LOC** to take its address. Otherwise, the object is assumed to be a local variable.
	INTEGER*4 *PROC* **EXTERNAL** *PROC* **INTEGER*4** *X* *X*=**LOCFAR**(*PROC*)	
C	**int** (******x*) ( );	---

When using C procedure pointers and calling a FORTRAN routine from C with the C calling convention, use the following syntax to declare the procedure pointers where you declare arguments for your C procedure:

returntype (***** *x*)(*typelist*)

The *returntype* is the C type of the return value. The *typelist* is given with the same syntax used to declare the argument list of a **fortran** routine from C. When using the FORTRAN calling convention, use the following syntax:

returntype (**fortran** * *x*)(*typelist*)

See Section 11.3.8.9 for information about using FORTRAN procedural arguments with C. FORTRAN procedural arguments are *not* compatible with C procedure pointers.

11.3.8.7 Arrays and Huge Arrays

FORTRAN arrays are allocated in column order. For example, `A(2,1)` is followed by `A(3,1)`. C arrays are allocated in row order. For example, `A(2,1)` is followed by `A(2,2)`.

The lower bound of indices to C arrays is always 0. For FORTRAN, the lower bound is 1 by default. However, you can specify a lower bound of 0 in the **DIMENSION** statement you use to declare an array.

For example, if you define a C array `x[6][3]`, an equivalent array in FORTRAN would be `X(3,6)`. If you specify element `x[5][0]` in C, the equivalent FORTRAN element is `X(1,6)`. Alternatively, you can define the equivalent FORTRAN array as `X(0:2,0:5)`, in which case the equivalent FORTRAN element would be `X(0,5)`.

If you define a FORTRAN array like this

```
INTEGER*2 X(2,5)
```

the equivalent C array is

```
short x[5][2]
```

In C, arrays are always passed by reference. If you declare an array in your FORTRAN program using the **C** attribute, the array is passed by value, like a C structure. That is, the entire array is laid out on the stack. To pass an array as an array from FORTRAN to C, you must declare it with the **REFERENCE** attribute in your FORTRAN program, or you must apply the **LOC**, **LOCNEAR**, or **LOCFAR** intrinsic function to the array and pass the result.

There are two methods for using C arrays of two or more dimensions in FORTRAN procedures:

1. Use the **typedef** statement to define a synonym, *name*, for the array *type* [*m*][*n*]..., as follows:

   ```
   typedef type name[m][n]...;
   ```

   Declare the FORTRAN procedure as

   ```
   extern void fortran f(name);
   ```

   In your main program, declare a variable of the type you have defined (*name*), then use that variable as the argument of the FORTRAN procedure, as follows:

   ```
   name x;
   f(x);
   ```

2. Declare the FORTRAN procedure as follows:

   ```
   extern void fortran f(type[m][n]...);
   ```

   In your main program, declare a variable as follows:

   ```
   type x[m][n];
   ```

   Then use that variable as the argument of the FORTRAN procedure as follows:

   ```
   f(x);
   ```

For example, when using the first method to pass a two-dimensional array, define the synonym `shortarraytype` as follows:

```
typedef short shortarraytype[2][2];
```

The type `shortarraytype` is now equivalent to `short[2][2]`. Now, declare the FORTRAN function `p` as follows:

```
extern void fortran p(shortarraytype);
```

In your main program, declare a variable `x` of type `shortarraytype`, then use `x` as the argument to the procedure `p` as follows:

```
main()
      {
      shortarraytype x;
      p(x);
      }
```

Table 11.19 shows equivalent array types for C and FORTRAN.

Table 11.19
Arrays

Language	Data Type	Notes
FORTRAN	*type X(M,J)*	---
C	*type x[j][m];*	When passed by reference
	struct { *type x[j][m];} x;*	When passed by value

11.3.8.8 Structures

In FORTRAN you can simulate a single instance of a structure by using the **EQUIVALENCE** statement, but there is no way to replicate the instance or apply such a structure to an argument. If the structure contains only fields of the same size, you can use an array. Otherwise, you need to define an equivalence group with variables associated in an **EQUIVALENCE** statement so that they map on to the appropriate elements of the structure.

If the whole structure is less than 32,767 bytes long, you can use a character variable to represent the whole structure. This means that you can assign a value with a single statement. This approach results in inefficient code and programs that are difficult to follow.

It is recommended that you use C to write interface procedures where possible. These could, for example, translate the structure into separate variables and scalars, which are easier to use with FORTRAN.

Use C structures to correspond to FORTRAN **COMPLEX** data types, as shown in Tables 11.20 and 11.21.

Table 11.20

Single-Precision Complex Numbers

Language	Data Type
FORTRAN	**COMPLEX** *X* **COMPLEX*8** *X*
C	**struct** *complex8* { **float** *re,im;*} *x;*

Table 11.21

Double-Precision Complex Numbers

Language	Data Type
FORTRAN	**COMPLEX*16** *X*
C	**struct** *complex16* { **double** *re,im;*} *x;*

C structures can also be used to pass FORTRAN logical values. For FORTRAN logical values, 1 means true and 0 means false. Tables 11.22 – 11.24 give examples of passing FORTRAN logical values.

Table 11.22

1-Byte Logical Values

Language	Data Type	Notes
FORTRAN	**LOGICAL*1** *X*	---
	LOGICAL	If **/4I2** or **$STORAGE:2** is in effect
C	**char** **x*;	Reference
	char *x*;	Value

Table 11.23

2-Byte Logical Values

Language	Data Type	Notes
FORTRAN	**LOGICAL*2** *X*	---
	LOGICAL	If **/4I2** or **$STORAGE:2** is in effect
C	**struct** { **char** *logical*; **char** *pad*[1]; } *x*;	---

Table 11.24

4-Byte Logical Values

Language	Data Type
FORTRAN	**LOGICAL*4** *X*
C	**struct** { **char** *logical*; **char** *pad*[3]; } *x*;

11.3.8.9 Procedural Parameters

Formal procedural arguments in FORTRAN are not compatible with procedure pointers in C.

However, FORTRAN procedural arguments can be represented by a C structure that mimics the FORTRAN sequence.

If you are calling C from FORTRAN, it is recommended that you use C procedure pointers. See Table 11.18 for equivalent procedure-pointer types.

11.3.9 Return Values

FORTRAN routines can return values to a C program. To write C programs that handle the return values correctly, you must understand the correspondence between data types in the different languages.

The C compiler performs conversions on return values before they are actually returned to the calling procedure. These conversions are the same as those given for arguments. Integral values that are shorter than an **int** are expanded to **int** size, and **float** values are converted to **double**. These types are discussed in Sections 11.3.8.2, "Integers," and 11.3.8.4, "Real Numbers."

The C compiler detects structured return values that are 4 bytes or less in length and returns them as integers of the appropriate size.

11.3.10 Sharing Data

FORTRAN common blocks are public data areas and can be referenced as external data objects in C. You can use the common-block names as the names of structure variables in C, for example. Blank common data have the public name **$COMMQQ**. FORTRAN cannot access C data objects without using the **EXTERN** attribute.

Alternatively, you can use the **LOC** intrinsic function in FORTRAN to give the address of a common block. Use **LOC** on the first variable in the common block, pass the address to a C procedure, and use that address from C. For example:

```
      INTERFACE TO SUBROUTINE CFUNC[C] (EXTP)
      INTEGER*4 EXTP
      END
      COMMON/EXT/I,J
      CALL CFUNC (LOC(I))
      .
      .
      .
      END
```

```
void cfunc (ext)
struct {long i,j;}*ext
       {
       ext->i = ext ->j;
       }
```

Or you can use the following method:

```
C When you have several common blocks to set up

      SUBROUTINE SETADS (ADSEXT, ADSPAR, ADSBL)
      INTEGER*4 ADSEXT,ADSPAR,ADSBL
      COMMON/EXT/I1
      COMMON/PAR/I2
      COMMON I3
      ADSEXT=LOCFAR(I1)
      ADSPAR=LOCFAR(I2)
      ADSBL=LOCFAR(I3)
      END

long *ext, *par, *blank;
void fortran setads( long **, long **, long **);
main( )
     {
     long formal;
     setads( &ext, &par, &blank);
     ext[0] = 100000;     /* Set FORTRAN common variable I1
to 100,000 */
     .
     .
     .
     }
```

11.3.11 Input and Output

A given file can be opened by only one language at a time, except for the standard output channel when that channel refers to the terminal. In this case, each FORTRAN **WRITE** statement that refers to the terminal should be followed by

```
WRITE(*,*)
```

if there is a possibility that a C routine might write to the terminal immediately thereafter. This clears the carriage-control character.

11.3.12 Run-Time Library Considerations

The FORTRAN run-time libraries **MLIBFOR*x*.LIB** and **LLIBFOR*x*.LIB** include the **system** routine and a subset of the **spawnlp** routine (as well as other routines) from the C library. FORTRAN programs can access these routines in the FORTRAN run-time libraries; however, these routines may present compatibility problems with future versions of Microsoft C. Sections 11.3.12.1 – 11.3.12.3 show how to access these C routines and discuss related

compatibility issues. (The demonstration program **DEMOEXEC.FOR** included with the Microsoft FORTRAN Compiler also gives examples of how to call these C routines.)

11.3.12.1 Accessing system and spawnlp Functions in FORTRAN Libraries

The C **system** function has the following declaration:

```
        int system(string)
    char *string;
```

The **system** function passes a specified C string (ending with a null character) to the DOS command interpreter (**COMMAND.COM**), which interprets and executes the string as a DOS command. This allows DOS commands (such as **DIR** or **DEL**), batch files, and programs to be executed.

The following program fragment shows how to access **system** from a FORTRAN program to display all files in the current directory with the file extension **.FOR**:

```
        INTEGER*2 SYSTEM

C   RETURN TYPE MUST BE DECLARED;
C   NOTICE THE C LITERAL STRING '...'C

        I = SYSTEM('DIR *.FOR'C)
```

The **INTERFACE** statement required to access **system** is given below. The **C** attribute is specified after the function name. The argument string includes the **REFERENCE** attribute to indicate that the argument is passed by reference.

```
       INTERFACE TO INTEGER*2 FUNCTION SYSTEM [C]
      +(STRING[REFERENCE])
       CHARACTER*1 STRING
       END
```

The C **spawnlp** function has the following declaration in C:

```
int spawnlp(mode,path,arg0,arg1,...,argn)
   int mode;    /* spawn mode */
   char *path; /* pathname of program to execute */
   char *arg0; /* should be the same as path*/
   char *arg1,...,*argn;    /* command line arguments */
   /* argn must be NULL */
```

This function creates and executes a new child process. There must be enough memory to load and execute the child process. The **mode** argument determines which form of **spawnlp** is executed. For the version of **spawnlp** in the FORTRAN run-time libraries, this value must be 0, which tells **spawnlp** to suspend the parent program and execute the child program. When the child program terminates, the parent program resumes execution. The return value from **spawnlp** is −1 if an error occurs. If the child process runs successfully, the return value is the return code from the child process.

The **path** argument specifies the file to be executed as the child process. The path can specify a full path name from the root directory, a partial path name from the current working directory, or just a file name. If the path argument does not have a filename extension or end with a period (.), the **spawnlp** call first appends the extension **.COM** and searches for the file. If **spawnlp** cannot find the file, it appends the extension **.EXE** and tries to find the file again. The **spawnlp** routine also searches for the file in any of the directories specified in the **PATH** environment variable using the same procedure.

The **INTERFACE** statement required to access **spawnlp** from a FORTRAN program is given below. The **C** attribute must appear after the function name. The **VARYING** attribute indicates that a variable number of arguments may be passed to the function.

```
      INTERFACE TO INTEGER*2 FUNCTION SPAWN
     + [C,VARYING,ALIAS:'_SPAWNLP'](MODE)
      INTEGER*2 MODE
      END
```

By default, you must name the routine SPAWNLP in the **INTERFACE** statement and use the **ALIAS** attribute to associate the FORTRAN name SPAWN with the C identifier **spawnlp**. You must use **ALIAS** in this case because the C name **spawnlp** has seven characters, but only the first six characters of FORTRAN names are significant. If you specify the **/4Nt** option when you compile (or use the **$NOTRUNCATE** metacommand in the source file), you can name the routine SPAWNLP in the **INTERFACE** statement.

The following program fragment illustrates how to call **spawnlp** from a FORTRAN program:

```
C   (THE RETURN TYPE MUST BE DECLARED)
      INTEGER*2 SPAWN
      ...
C   EXECUTE AS A CHILD PROCESS

      I = SPAWN(0, LOC('EXEMOD'C), LOC('EXEMOD'C), c
     +      LOC('DEMOEXEC.EXE'C), INT4(0))
```

Notice in this example that the method used to pass strings to **spawnlp** is different from the method used to pass strings to **system**. This is because the string arguments to **spawnlp** are undeclared in the **INTERFACE** statement and assumed to be passed by value. The **spawnlp** function expects the addresses of the strings rather than the actual characters, so the FORTRAN program uses the **LOC** intrinsic function to pass the address. (Remember that functions with the **C** attribute pass arguments by value). The last argument to the **spawnlp** routine must be a C null pointer (an integer 0), so the FORTRAN program must use either the `INT2(0)` or the `INT4(0)` intrinsic function (depending on the memory model) to pass this pointer by value as the last argument.

11.3.12.2 FORTRAN Libraries and Future Versions of Microsoft C

If you plan to mix Microsoft FORTRAN modules with modules compiled using Microsoft C, you should use the **SETUP** utility to prepare new copies of your FORTRAN run-time libraries. When you create these copies, tell **SETUP** that you want the copies to interface with C. In these copies, **SETUP** removes the C routines. Later, when you link FORTRAN modules with C modules, you will need to link with the appropriate standard C run-time library for the memory model you are using; this will take care of unresolved references from **MLIBFOR*x*.LIB** and **LLIBFOR*x*.LIB**. However, you can still use your original copies of **MLIBFOR*x*.LIB** and **LLIBFOR*x*.LIB** for any programs that do not include C modules.

11.3.12.3 Linking Considerations

When you link modules compiled with Microsoft FORTRAN and Microsoft C, the linker options you use and the order in which you link the modules are both significant. Linking FORTRAN and C modules imposes the following requirements:

- You must compile and link in separate steps; that is, after compiling, you must use the **LINK** command to invoke the linker explicitly.
- You must have compiled all of the object modules you are linking with the same floating-point and memory-model options.
- On the **LINK** command line, you must specify the **/NOD** option and you must explicitly give the libraries you are using for linking and the order in which they should be searched.

Linking Version 4.0 FORTRAN and C Modules

A **LINK** command of the following form is required to link modules compiled with Version 4.0 of FORTRAN and C:

LINK /NOD *objfile*[,*objfile*...],,,{**L** | **M**}**LIBFOR**{**E** | **7** |**A**}+{**L** | **M**}**LIBC**

The FORTRAN library and the C library must be for the same memory model. Also, the FORTRAN library must include compatibility with C (as specified during **SETUP**; see Section 2.4.6.5). The FORTRAN library must appear before the C library on the **LINK** command line.

Linking Version 3.3 or Version 4.0 FORTRAN Modules with C Modules

If you are linking modules compiled with Version 3.3 of FORTRAN, plus modules compiled with Version 4.0 of FORTRAN, plus modules compiled with C, you must use a **LINK** command of the following form:

LINK /NOD *objfile*[,*objfile*...],,,**LLIBFOR**{**E** | **7** | **A**}+**LLIBC**+**FORTRAN**

The FORTRAN library must include compatibility with C and with Version 3.3 (as specified during **SETUP**; see Section 2.4.6.6 for more information). You must give the Version 4.0 FORTRAN library before the C library on the **LINK** command line, and you must give the FORTRAN compatibility library last on the command line so that it is the last library to be searched.

Note that you must use the large memory model, since Version 3.3 of FORTRAN does not support other memory models.

11.3.13 Error Messages

If errors occur during compilation, the compiler for the language that caused the error generates the error message.

Most run-time errors also come from the language in which the part of the program causing the error was written. However, floating-point errors caused by the FORTRAN libraries have the form of FORTRAN error messages, even if a C routine called the intrinsic function that caused the error. For FORTRAN, run-time error messages and floating-point error messages are identical; for C, floating-point error messages are slightly different from run-time error messages.

Appendixes

Appendixes

A Differences between Versions 4.0 and 3.3 303

B Using Exit Codes 335

C Microsoft FORTRAN Record and File Formats 343

D Handling 8087/80287 Floating-Point Exceptions 355

E Error Messages and Limits 365

Appendix A
Differences between Versions 4.0 and 3.3

A.1 Introduction 307
A.2 Changes for ANSI Full-Language Standard 307
A.3 Source Compatibility 309
A.3.1 Attributes in Array Declarations 309
A.3.2 Blanks in Formatted Files 309
A.3.3 MODE and STATUS Options in OPEN Statement 310
A.3.4 Temporary Scratch-File Names 310
A.3.5 Binary Direct Files 311
A.3.6 Precision of Floating-Point Operations 311
A.3.7 Exponentiation Exceptions 312
A.3.8 List-Directed Output 314
A.3.9 DO-Loop Ranges 315
A.4 Object Compatibility 315
A.4.1 Library Compatibility 315
A.4.2 Mixing Version 4.0 and Version 3.3 Modules 316
A.4.3 Mixing Version 4.0 and Version 3.2 Modules 317
A.5 Changes for Version 4.0 317
A.5.1 Enhancements and Additions to the Compiler and Linker 318
A.5.1.1 The FL Command 318
A.5.1.2 Changes to the Linker 319
A.5.1.3 Memory Models 319

A.5.2 Run-Time Library Changes 320
A.5.3 Changes to the Language 320
A.5.3.1 Underscore (_) as a Digit 321
A.5.3.2 Dollar Sign ($) in Collating Sequence 321
A.5.3.3 Significant Characters in Names 321
A.5.3.4 Column Restrictions for Source Files 322
A.5.3.5 Restrictions on Continuation Lines 322
A.5.3.6 Maximum Character-Value Length 322
A.5.3.7 Arithmetic Operations 322
A.5.3.8 Character Editing and Hollerith Data Types 323
A.5.3.9 Expressions in Substring Specifications 323
A.5.3.10 Array Subscripts 323
A.5.3.11 Changes to the Input/Output System 324
A.5.3.12 Assignment Statement (Computational) 325
A.5.3.13 CALL Statement 325
A.5.3.14 DATA Statement 326
A.5.3.15 BACKSPACE, ENDFILE, and REWIND Statements 326
A.5.3.16 CLOSE and OPEN Statements 326
A.5.3.17 DIMENSION Statement 326
A.5.3.18 DO Statement 327
A.5.3.19 INQUIRE Statement 328
A.5.3.20 PAUSE Statement 328
A.5.3.21 READ and WRITE Statements 328
A.5.3.22 STOP Statement 329
A.5.3.23 Type Statements 329
A.5.3.24 Conditional Compilation 329

A.5.4 New Language Features 330
A.5.4.1 INTEGER*1 and LOGICAL*1 Data Types 330
A.5.4.2 C Strings 330
A.5.4.3 Concatenation Operator 331
A.5.4.4 New Intrinsic Functions 331
A.5.4.5 New Time and Date Functions 333
A.5.4.6 Z Edit Descriptor 333
A.5.4.7 ENTRY Statement 333
A.5.4.8 PRINT Statement 333
A.5.4.9 $[NO]DECLARE, $[NO]FREEFORM, and $[NO]TRUNCATE Metacommands 334

A.1 Introduction

This appendix describes features of the Microsoft FORTRAN Compiler, Version 4.0, that are extensions of or changes to Version 3.3. It summarizes the changes made in Version 4.0 to support the ANSI full-language standard; discusses compatibility between source and object files for Versions 3.2, 3.3, and 4.0; and describes changes and additions to the compiler and linker software, the run-time library system, and the language itself.

Important

You must use the **SETUP** program to create libraries that you link with Version 4.0 programs. Note that if you choose all the default responses for **SETUP**, the library that **SETUP** builds requires that you have an 8087 or 80287 coprocessor installed. (See Chapter 8 for more information about the compiler options and libraries used to control floating-point math.)

A.2 Changes for ANSI Full-Language Standard

Version 4.0 of the Microsoft FORTRAN Compiler is an implementation of the ANSI X3.9-1978 FORTRAN full-language standard; Version 3.3 implemented only the subset standard. The following list summarizes the new features in Version 4.0 that were required for the ANSI full-language standard.

Language Construct	**Change for Version 4.0**
Concatenation operator (**//**)	Now supported.
Asterisk length specifiers	Can be used with character functions and character parameters.
CHARACTER* *n* arguments	Argument length passed with **CHARACTER*** *n* arguments to subprograms or functions. The maximum value of *n* is now 32,767 instead of 127.

Format specifiers	Can be character arrays.
Unit specifiers	Unit specifiers that include the **UNIT=** keyword can appear at any position in the I/O control list.
LEN intrinsic function	Now supported.
INDEX intrinsic function	Now supported.
Assignment statement (computational)	Can include Hollerith constants.
BACKSPACE, **ENDFILE**, and **REWIND** statements	Can transfer control to a label after errors and use a variable to indicate error or end-of-file status.
CLOSE and **OPEN** statements	Can transfer control to a label after errors. The **OPEN** statement can specify how blanks are interpreted in numeric input.
DATA statement	Items that are assigned values can include substring names and implied-**DO** lists.
CALL statement	Can include Hollerith constants.
DATA statement	Can include Hollerith constants.
DIMENSION statement	Both upper and lower bounds allowed for dimension declarators.
DO statement	Loop indices can be of any **REAL** type.
ENTRY statement	Now supported.
STATUS= in **OPEN** statements	Opening existing files with the **STATUS='NEW'** option is illegal. **STATUS='UNKNOWN'** in Version 4.0 behaves the same way as does **STATUS='NEW'** in Version 3.3.
Constants in **PARAMETER** statements	Arithmetic, logical, and character constants fully supported.
PRINT statement	Now supported.
READ statement	**READ** statements without a control information list or without a unit specifier supported.

A.3 Source Compatibility

Version 4.0 of the Microsoft FORTRAN Compiler compiles any valid source program that you successfully compiled using an earlier version of the compiler, except where list-directed I/O and direct-access I/O are used together. However, source programs may behave differently when compiled with Version 4.0. See Sections A.3.1 – A.3.9 for a description of these behavior differences.

A.3.1 Attributes in Array Declarations

In array declarations in Version 4.0, attributes appear before the list of array bounds. In Version 3.3, attributes appear after the list of array bounds.

For example, this declaration in a Version 3.3 source file

```
DIMENSION x(10)[VALUE]
```

should appear as shown below in a Version 4.0 source file:

```
DIMENSION x[VALUE](10)
```

A.3.2 Blanks in Formatted Files

The ANSI full-language and subset standards treat blanks in formatted files differently. In the full-language standard, blanks are treated as null characters unless the **BN** and **BZ** format descriptors, or the **BLANK=** option in an **OPEN** statement, specify otherwise. In the subset standard, blanks are treated as zeros unless the **BN** and **BZ** format descriptors indicate otherwise.

Version 4.0 supports the full-language treatment of blanks: it considers blanks to be null characters unless otherwise specified.

If the files used by a program expect blanks to be treated as zeros by default, the program must include the **BLANK='ZERO'** option in the **OPEN** statements for those files.

A.3.3 MODE and STATUS Options in OPEN Statement

In Version 4.0, if the **MODE=***mode* option does not appear in an **OPEN** statement, the FORTRAN run-time system tries to open the file with **MODE** values of **'READWRITE'**, **'READ'**, and **'WRITE'**, in that order. In Version 3.3, if the **MODE=***mode* option does not appear in an **OPEN** statement, the FORTRAN run-time system tries to open the file with **MODE** values of **'READWRITE'**, **'WRITE'**, and **'READ'**, in that order.

In Version 4.0, when the **STATUS='NEW'** option appears in an **OPEN** statement, the file specified in the statement must not exist. If an existing file has the same path name as the file specified in the statement, an error results. In Version 3.3, when the **STATUS='NEW'** option appears in an **OPEN** statement, the file specified in the statement can exist at the time the statement is executed. Any file with the same path name as the file specified in the statement is overwritten. (This conflicts with a strict interpretation of the standard.)

If you want programs compiled using Version 4.0 to behave in the same way as programs compiled using Version 3.3, substitute the **STATUS='UNKNOWN'** option for the **STATUS='NEW'** option in any **OPEN** statements that specify the path names of existing files.

A.3.4 Temporary Scratch-File Names

In Version 4.0, if no file name is specified in an **OPEN** statement, the FORTRAN run-time system creates a temporary "scratch" file with a file name in the following format:

ZZ*processno*

In this file name, *processno* is an alphanumeric character followed by a 5-digit process number. The alphanumeric character is "0" for the first temporary file opened, followed by the letters "a", "b", "c", and so on for each subsequent file name. For example, if you opened five files with no file names in a single program, the file names assigned to the temporary files would be the following (if "12345" is the process number):

```
ZZ012345 (first file opened)
ZZa12345 (second file opened)
ZZb12345 (third file opened)
ZZc12345 (fourth file opened)
ZZd12345 (fifth file opened)
```

In Version 3.3, if no file name is specified in an **OPEN** statement, the "scratch" file name has the following format:

T*unitspec*.**TMP**

In this file name, *unitspec* is the unit number specified in the **OPEN** statement.

A.3.5 Binary Direct Files

In Version 4.0, binary files can be opened for direct access. In most cases, I/O operations performed on binary direct files produce the same results as the same operations performed on unformatted direct files. An exception is that the number of bytes transferred in a single binary direct read or write operation is no longer limited by the record length (although even multiples of the record length are still used in repositioning between successive **READ** and **WRITE** statements).

See Appendix C of this manual and Sections 4.3.3.2, 4.3.4, and 4.4 of the *Microsoft FORTRAN Compiler Language Reference* for more information about binary direct files.

A.3.6 Precision of Floating-Point Operations

Programs that use floating-point values may give slightly different results when compiled with Version 4.0 because Version 4.0 passes more information to the 8087/80287 coprocessor than Version 3.3. This has the effect of maintaining higher precision than if the values were truncated into double- or single-precision values.

For example, in Version 4.0, arguments to transcendental functions are passed in the 8087/80287 registers. If these arguments are expressions, their values are in the 64-bit precision of the coprocessor. In Version 3.3, arguments to transcendental functions are passed in memory as either single- or double-precision values. Thus, these arguments are truncated to 23- or 52-bit precision, respectively.

See Section 3.3.15, "Optimizing," for a discussion of the **/Op** option, and Chapter 8, "Controlling Floating-Point Operations," for more information about floating-point operations.

A.3.7 Exponentiation Exceptions

Versions 4.0 and 3.3 give different results for certain cases of exponentiation. These differences fall into four categories:

1. Zero raised to a zero power
2. Zero raised to a negative power
3. **COMPLEX** zero raised to a **COMPLEX** power
4. Negative **INTEGER** or **REAL** values raised to a **REAL** power

Tables A.1 – A.4 summarize these differences. The following abbreviations are used in the tables:

Abbreviation	Meaning
$-n$	Negative integer
$-r$	Negative real number
$+r$	Positive real number
s	Nonzero real number
w	Integral real number (for example, 3.0)
f	Nonintegral real number (for example, 1.5)

Table A.1

Negative INTEGER or REAL Raised to a REAL Power

Base Type	Exponent Type	Formula	Example	Version 4.0 Returns	Version 3.3 Returns
INTEGER	**REAL**	$(-n)^w$	$(-3)^{3.0}$	-27.0	Error†
INTEGER	**REAL**	$(-n)^f$	$(-1)^{1.5}$	Error	Error†
REAL	**REAL**	$(-r)^w$	$(-3.0)^{3.0}$	-27.0	Error†
REAL	**REAL**	$(-r)^f$	$(-1.0)^{1.5}$	Error	Error†

† Version 3.3 does not allow exponentiation of a negative number to a **REAL** power. Version 4.0 allows it only if the exponent is a whole number, such as 3.0; it does not allow fractional exponents such as 1.5. These restrictions do not apply to exponentiation with a **COMPLEX** base (or exponent); for example, `COMPLEX (-1.0,0.0)**1.5` will give (0.0, – 1.0) as the result.

Table A.2
Zero Raised to a Negative Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
INTEGER	**INTEGER**	0^{-n}	Error	Error
REAL	**INTEGER**	0.0^{-n}	Error	Error†
REAL	**REAL**	0.0^{-r}	Error	Error†
COMPLEX	**INTEGER**	$(0.0,0.0)^{-n}$	Error	(0.0,0.0)
COMPLEX	**REAL**	$(0.0,0.0)^{-r}$	Error	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(-r,0)}$	Error	(0.0,0.0)

† In Version 3.3, **REAL** 0.0 raised to a negative power produces an error if exceptions are not masked with **LCWRQQ**, and infinity if exceptions are masked with **LCWRQQ**.

Table A.3
COMPLEX Zero Raised to a COMPLEX Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
COMPLEX	**COMPLEX**	$(0.0, 0.0)^{(+r,0.0)}$	(0.0,0.0)	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(0.0,0.0)}$	(1.0,0.0)	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(-r,0.0)}$	Error	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(+r,s)}$	(0.0,0.0)	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(0.0,s)}$	Error	(0.0,0.0)
COMPLEX	**COMPLEX**	$(0.0,0.0)^{(-r,s)}$	Error	(0.0,0.0)

Table A.4

Zero Raised to the Zero Power

Base Type	Exponent Type	Formula	Version 4.0 Returns	Version 3.3 Returns
INTEGER	INTEGER	0^0	1	1
REAL	INTEGER	0.0^0	1.0	1.0
REAL	REAL	$0.0^{0.0}$	1.0	1.0
COMPLEX	INTEGER	$(0.0,0.0)^0$	(1.0,0.0)	(0.0,0.0)
COMPLEX	REAL	$(0.0,0.0)^{0.0}$	(1.0,0.0)	(0.0,0.0)
COMPLEX	COMPLEX	$(0.0,0.0)^{(0.0,0.0)}$	(1.0,0.0)	(0.0,0.0)

A.3.8 List-Directed Output

In Version 4.0, the conventions for list-directed output have changed. The following conventions are used:

1. Integer output constants are produced with the effect of an **I11** edit descriptor. (Version 3.3 uses the **I12** edit descriptor for this.)

2. Real and double-precision constants are produced with the effect of either an **F** or an **E** edit descriptor, depending on the value of the constant *c* in the following range:

   $$1 <= c < 10^7$$

   a. If *c* is within the range, the constant is produced by using `0PF15.6` for single precision and `0PF24.15` for double precision. In Version 3.3, `0PF16.7` is used for single precision and `0PF23.14` is used for double precision.

   b. If *c* is outside the range, the constant is produced using `1PE15.6E2` for single precision and `1PE24.15E3` for double precision. The value 0 is printed with this format. (In Version 3.3, `1PE14.6E3` is used for single precision and `1PE21.13E3` is used for double precision.)

   The same field widths are used to force the constants in both cases to line up on a printed page.

A.3.9 DO-Loop Ranges

The code generated for **DO** loops in Version 4.0 uses the standard formula for determining the loop iteration count, which is, consequently, limited to the maximum allowable integer size. In Version 3.3, the code generated for **DO** loops allows more iterations than the maximum allowable integer value; for example, if the **$STORAGE:2** metacommand is in effect, a **DO** loop of the following form loops 65,535 times in Version 3.3 but is illegal in Version 4.0:

```
DO 200 I = -32767,32767
```

A.4 Object Compatibility

Sections A.4.1 – A.4.3 discuss compatibility between object files compiled with Versions 4.0, 3.3, and 3.2. If possible, you should recompile programs and subprograms to take advantage of the improved code generated by Version 4.0. If you cannot do this (for example, if the source files are unavailable), you can continue to link object files generated by Version 3.3 with those generated by Version 4.0. However, you should read the information in the following paragraphs to make sure that object files compiled under the two versions link correctly.

A.4.1 Library Compatibility

If your program mixes modules compiled with Version 4.0 and modules compiled with Version 3.3, you must link them with the **FORTRAN.LIB** library that comes with Version 4.0 in addition to a standard FORTRAN library built by the **SETUP** program. The **SETUP** program installs the Version 4.0 **FORTRAN.LIB** if you request compatibility with Version 3.3 or 3.2. This library is required because the standard Version 4.0 libraries are different internally from the standard Version 3.3 and Version 3.2 libraries, and the code generated by the Version 4.0 compiler accesses these libraries differently. Thus, special interfaces are required so that the code produced by the two versions can work together.

The Version 4.0 **FORTRAN.LIB** library includes the interfaces required to work with Version 3.3 and Version 3.2 modules. It contains all the external interfaces supported by Version 3.3 and Version 3.2 **FORTRAN.LIB**. However, the interfaces in the Version 4.0 library generally use parts of the standard Version 4.0 library to perform their processing.

FORTRAN.LIB is not required if all of the object files you are linking were compiled with Version 4.0. Also, since modules compiled with Versions 3.3 and 3.2 have library search directives for **FORTRAN.LIB** embedded in them, you do not need to specify **FORTRAN.LIB** explicitly when you link. However, this library should be found in the standard place specified in the **LIB** environment variable.

You can use Version 3.3 and Version 3.2 modules with Version 4.0 modules that are compiled with any **/FP** compiler option, subject to the restrictions that apply to the Version 4.0 modules: that is, you cannot link with an alternate math library (**LLIBFORA.LIB**) if any of the modules contains in-line instructions. However, you must still tell the **SETUP** program to include the "compatibility" math interfaces in the **LLIBFOR*x*.LIB** library that it builds if you plan to use the library with Version 3.3 and Version 3.2 modules. The resulting program will not be affected, but the library that **SETUP** builds will be slightly larger. (The math interfaces are not included in **FORTRAN.LIB** since, unlike the standard FORTRAN libraries built by **SETUP**, **FORTRAN.LIB** is not typically associated with a particular **/FP** option.)

A.4.2 Mixing Version 4.0 and Version 3.3 Modules

Version 4.0 modules that are linked with Version 3.3 modules must be compiled using the large memory model. This model is the default for Version 4.0 FORTRAN programs. (See Chapter 9 for more information about memory models.)

In most cases, the calling and argument-passing conventions are the same in Versions 3.3 and 4.0, so that routines compiled under either version can call each other freely. The only exception is the case of a Version 3.3 routine calling a Version 4.0 routine and passing a **CHARACTER*(*)** argument. (This situation is most likely to arise when a Version 3.3 program passes a subprogram as an argument to another subprogram compiled with Version 4.0.)

A routine compiled with Version 3.3 cannot call a Version 4.0 routine that has **CHARACTER*(*)** formal arguments. Version 4.0 expects the caller to specify the lengths of all such arguments in a special way. Since Version 3.3 does not support arguments of this type, Version 3.3 programs cannot pass the argument length. Any such call gives undefined results at run time. (This change was made in order to support the more powerful feature of the full ANSI FORTRAN-77 standard.) This problem does not arise in calls from Version 4.0 routines to Version 3.3 routines. Version 4.0 routines pass the length of a **CHARACTER*(*)** argument in such a way that Version 3.3 routines can safely ignore it.

Note

Certain additional rules apply if you are linking C modules with FORTRAN modules. Section 11.3.12.3 explains these rules.

If you compile a Version 3.3 source file that includes the **STATUS='NEW'** option and link the resulting object file with a Version 4.0 library that includes the Version 3.3 compatibility package, the **STATUS='NEW'** option is mapped to **STATUS='UNKNOWN'**. This results in behavior more similar to the Version 3.3 implementation of the **STATUS='NEW'** option.

A.4.3 Mixing Version 4.0 and Version 3.2 Modules

In general, programs can mix modules compiled with Versions 4.0 and 3.2 of Microsoft FORTRAN. However, the following considerations apply:

- All considerations that apply to mixing Version 3.3 modules with Version 4.0 modules also apply to mixing Version 3.2 modules with Version 4.0 modules. (See Sections A.4.1 and A.4.2 for more information.)
- You must compile any Version 4.0 modules in these programs with the **/Gr** option to the **FL** command. This is because the code that Version 4.0 generates by default preserves the **SI** and **DI** registers for the duration of a subprogram, while the code that Version 3.2 generates does not. If you specify **/Gr**, the Version 4.0 code does not expect the **SI** and **DI** registers to be preserved.

A.5 Changes for Version 4.0

Sections A.5.1 – A.5.4 discuss changes and enhancements to the Microsoft FORTRAN Compiler for Version 4.0. These changes fall under the following categories:

- Enhancements and additions to the compiler and linker
- Run-time library changes
- Language changes

A.5.1 Enhancements and Additions to the Compiler and Linker

Several features have been added to, or changed in, Version 4.0 of the Microsoft FORTRAN Compiler and the Microsoft Overlay Linker (**LINK**) to make them easier to use. These features should not affect your source code, but you may need to revise existing batch files or **MAKE** description files so that they work correctly with Version 4.0.

A.5.1.1 The FL Command

In Microsoft FORTRAN, a new command, **FL**, automatically executes the compiler and the linker. The options associated with this command give you considerable flexibility in controlling compilation and linking.

You can specify the **/c** option with the **FL** command to compile without linking. You can invoke the linker separately after you compile, either through **FL** or through the **LINK** command. See Section 3.4 for information on how to use the **FL** command to link without compiling; see Chapter 4 for a description of the use of the **LINK** command and its options.

The **FL** command performs many of the same functions as any batch files that you may have created to compile and link your FORTRAN programs. It also allows you to specify on the command line all files you want to compile and link and all options for controlling the process. You can include wild-card characters in the files you specify, so that you can easily compile and link more than one file. **FL** automatically prompts you if it cannot find a file that it needs at any point during compilation and linking. Note that you must give the entire source-file name, including the **.FOR** extension, to the **FL** command. If you do not include the **.FOR** extension, **.FL** interprets the file name as an object-file name.

If you wish to convert existing batch files so that they compile and link correctly under Version 4.0, be sure that you substitute the appropriate **FL** command for any **FOR1**, **PAS2**, **PAS3**, and **LINK** commands that may have been in the original batch files.

See Chapter 3, "Compiling: The FL Command," for detailed instructions on using the **FL** command for program compilation and linking.

A.5.1.2 Changes to the Linker

Several linker options have been added for Version 4.0. You can specify these options either by using the **/link** option of the **FL** command, or by using the **LINK** command if you choose to invoke the linker separately.

The following list gives the new linker options:

Option	Task
/CO	Prepares a special executable file for use with the Microsoft CodeView window-oriented debugger
/DO	Enforces the default segment-loading order for Microsoft language products (including Microsoft FORTRAN)
/E	Packs the executable file during linking
/HE	Lists all **LINK** command options on standard output
/I	Displays information about the effect of the linking process on standard error output

See Sections 4.6.1 – 4.6.17 for information about how to use these options.

A.5.1.3 Memory Models

When you compile a program using Version 4.0 of the Microsoft FORTRAN Compiler, you can choose a memory model to be used for your program. The memory model you choose specifies how memory for the code and data in your program will be allocated. Three memory models are available: medium, large, and huge. You choose a memory model by specifying the **/AL** (large), **/AM** (medium), or **/AH** (huge) option with the **FL** command at compile time. The default is the large memory model. (See Chapter 9 for information on the use of memory models.)

All programs compiled with Version 3.3 of the Microsoft FORTRAN Compiler are large-model programs. The large model is the default memory model for Version 4.0.

For programs that mix modules compiled under Versions 3.3 and 4.0, Version 4.0 modules cannot be compiled using the medium memory model. If this model is used for the Version 4.0 modules, the program may produce undefined results, although it may appear to link correctly.

Note

Using the **$LARGE** metacommand on an entire program has the same effect as specifying the huge memory model, except that fixed-size arrays are implicitly declared with the **HUGE** attribute. The **$LARGE** metacommand is not associated with the large memory model.

A.5.2 Run-Time Library Changes

The following changes have been made to the libraries provided with Version 4.0 of the Microsoft FORTRAN Compiler:

- The auxiliary library **DECMATH.LIB**, which supported an alternative floating-point format in Version 3.3, is no longer provided.
- The library structure for Version 4.0 is considerably different from the structure for Version 3.3. During installation, you can specify the memory model, the math package you wish to use, and various other options. Then the **SETUP** program builds a library according to your specifications. (See Chapter 2, "Getting Started," for more information about how libraries are built during the installation process.) The memory-model and floating-point options you specify on the **FL** command line allow your program to be linked with the library you build automatically. (Section 3.3.1 shows which library is used for each combination of floating-point and memory-model options.)

A.5.3 Changes to the Language

This section lists the changes made to the Microsoft FORTRAN language for Version 4.0. For each difference, a reference to the appropriate section in the documentation for Version 4.0 or Version 3.3 is given. Section numbers from the *Microsoft FORTRAN Compiler Language Reference* are preceded by "LR"; section numbers from the *Microsoft FORTRAN Compiler User's Guide* are preceded by "UG."

A.5.3.1 Underscore (_) as a Digit

In Version 4.0, the underscore is classified as a digit, which can be used as any character of a name other than the first character. An underscore cannot be used in names if the **/4Ys** option is used in compiling (or the **$STRICT** metacommand is in effect). In Version 3.3, the underscore (_) is classified as a special character, which cannot be used in names.

A.5.3.2 Dollar Sign ($) in Collating Sequence

In Version 4.0, the dollar sign is classified as an alphanumeric character, which can be used in names and which appears after uppercase Z in the collating sequence (LR:2.2).

The dollar sign cannot be used as an alphanumeric character in names in the following cases:

- If the **/4Ys** option is used in compiling (or the **$STRICT** metacommand is in effect)
- If the name is declared using the **C** attribute

In Version 3.3, the dollar sign ($) is classified as a special character, which appears as the first character in the FORTRAN collating sequence (LR:2.1).

A.5.3.3 Significant Characters in Names

In Version 4.0, only the first six characters in a name are significant, unless the **/4Nt** option is used in compiling or the **$NOTRUNCATE** metacommand is in effect (LR:2.3). In this case, the first 31 characters in a name are significant.

In Version 3.3, only the first six characters in a name are significant under any circumstances (LR:1.6).

A.5.3.4 Column Restrictions for Source Files

Version 4.0 allows source code to be in free-form format. The **/4Yf** option to the **FL** command (and the **$FREEFORM** metacommand) gives you this choice (LR:3.4); see the description of the **$FREEFORM** metacommand in Section 6.2.5 of the *Microsoft FORTRAN Compiler Language Reference* for the rules that apply to free-form source files.

In Version 3.3, statements in source programs are required to obey the standard FORTRAN column restrictions (LR:2.1.4).

A.5.3.5 Restrictions on Continuation Lines

In Version 4.0, limits on the number of continuation lines have been removed, unless the **/4Ys** option is used in compiling (or the **$STRICT** metacommand is in effect). In these cases, the compiler generates an error if a statement extends over more than 19 continuation lines or includes more than 1320 characters (LR:3.2).

In Version 3.3, these restrictions are always in effect (LR:2.2.2).

A.5.3.6 Maximum Character-Value Length

In Version 4.0, the maximum length of character values is 32,767 characters (LR:2.4.6). Character constants are effectively limited to 1958 characters.

In Version 3.3, character values can have a maximum length of 127 characters (LR:2.3.6).

A.5.3.7 Arithmetic Operations

In Version 4.0, raising a negative-value operand to an integral real power is permitted.

In Version 3.3, raising a negative-value operand to any real power produces an error.

A.5.3.8 Character Editing and Hollerith Data Types

In Version 4.0, Hollerith data types can be used with the **A** edit descriptor when an input/output list item is of type **INTEGER**, **REAL**, or **LOGICAL** (LR:4.8.2.8).

A.5.3.9 Expressions in Substring Specifications

In Version 4.0, any type of arithmetic expression can be used to specify the first and last characters in a substring, unless the **/4Ys** is used in compiling (or the **$STRICT** metacommand is specified). In effect, noninteger substring expressions are truncated by an implicit use of the **INT** intrinsic function before substring operations are performed. If the **/4Ys** option (or **$STRICT** metacommand) appears, only integer expressions can be used to specify the first and last characters in a substring.

In Version 3.3, these restrictions are always in effect (UG:A.4).

In Version 4.0 the compiler verifies the following relationships, where *first* is the arithmetic expression that defines the first character in the substring, *last* is the arithmetic expression that defines the last character, and *length* is the length of the character variable:

- *first*<=*last*
- 1<=*first*<=*length*
- 1<=*last*<=*length*

If either of these relationships is false and the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect), the compiler generates an error message. If either of these relationships is false and the **/4Yb** option is not used (or the **$DEBUG** metacommand is not in effect), the substring is undefined (LR:2.4.6.2).

A.5.3.10 Array Subscripts

In Version 4.0, array subscripts can be any arithmetic expression, unless the **/4Ys** option is used in compiling (or the **$STRICT** metacommand is specified). In effect, noninteger subscript expressions are truncated by an implicit use of the **INT** intrinsic function before subscripting operations are performed. If the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is specified), subscripts are checked on all arrays that are not formal arguments, and an error message is generated for invalid subscripts (LR:2.5).

In Version 3.3, array-element references must be integer expressions (LR:2.5.9).

A.5.3.11 Changes to the Input/Output System

This section describes changes to the input/output system used in Version 4.0 of Microsoft FORTRAN.

Unit Specifiers

In Version 4.0, unit specifiers can be used more flexibly. The optional **UNIT=** string can appear before the unit specifier in all I/O statements except **PRINT**, **INQUIRE** with a **FILE=** option, and the **EOF** intrinsic function. If the optional **UNIT=** string appears in the unit specifier, the specifier can appear at any position in the I/O control list. This change was made to conform with the ANSI full-language standard for FORTRAN.

In Version 3.3, the unit specifier must appear in the first position.

In Version 4.0, the following external unit specifiers can be reconnected to another file:

External Unit	Description
0	Initially represents the keyboard or the screen
5	Initially represents the keyboard
6	Initially represents the screen

If you connect any of these specifiers to a different file using an **OPEN** statement and then close that file, the specifier resumes its preconnected status.

Output Lists

In Version 4.0, arbitrary expressions used in an output list can begin with a left parenthesis (LR:4.3.8).

In Version 3.3, arbitrary expressions used in an output list cannot begin with a left parenthesis because left parentheses are reserved for implied-**DO** lists (LR:4.3.1.3).

Format Specifiers

In Version 4.0, statement labels, integer variables, character expressions, character variables, or character arrays can be used as format specifiers. If the **/4Ys** option is not used in compiling (and the **$STRICT** metacommand is not in effect), noncharacter arrays can also be used (LR:4.3.7).

In Version 3.3, only statement labels, integer variables, character expressions, or character variables can be used as format specifiers (LR:4.3.1.2).

Backslash (\) Edit Descriptor

In Version 4.0, the backslash (\) edit descriptor is only recognized for files connected to terminal devices such as screens or printers. Otherwise, it is ignored (LR:4.8.1.7).

In Version 3.3, the backslash (\) edit descriptor is recognized for all file types (LR:4.4.2.1).

A.5.3.12 Assignment Statement (Computational)

In Version 4.0, the *expression* in a computational assignment statement can be a Hollerith constant. A Hollerith constant can be assigned to any type of *variable*. The normal rules for padding and truncation of character data types also apply to Hollerith constants.

In Version 3.3, Hollerith constants cannot be used in assignments.

A.5.3.13 CALL Statement

In Version 4.0, the *actuals* parameter can include Hollerith constants. Hollerith constants cannot be passed to character formal arguments (LR:5.3.5).

In Version 3.3, Hollerith constants cannot be used in **CALL** statements.

A.5.3.14 DATA Statement

In Version 4.0, the *nlist* parameter in a **DATA** statement can include substring names and implied-**DO** lists, and the *clist* parameter can include Hollerith constants. The normal rules for padding and truncation of character data types also apply to Hollerith constants (LR:5.3.11).

In Version 3.3, these constructs are not allowed (LR:3.2.9).

A.5.3.15 BACKSPACE, ENDFILE, and REWIND Statements

In Version 4.0, the **BACKSPACE**, **ENDFILE**, and **REWIND** statements can include an **ERR=** option to specify the flow of control after errors, and an **IOSTAT=** option to specify a variable to be used to indicate error or end-of-file status (LR:5.3.3, 5.3.18, 5.3.46).

In Version 3.3, the only option allowed in the **BACKSPACE**, **ENDFILE**, and **REWIND** statements is a unit specifier, which specifies the unit location of the file that the command acts on (LR:3.2.3, 3.2.15, 3.2.36).

A.5.3.16 CLOSE and OPEN Statements

In Version 4.0, the **CLOSE** and **OPEN** statements can include the **ERR=** option to specify the flow of control if an error occurs during statement execution (LR:5.3.7, 5.3.38). In addition, the **OPEN** statement can include the **BLANK=** option to indicate how blanks are interpreted in numeric input and the **BLOCKSIZE=** option to assign a new I/O-buffer size for the file being opened (LR:5.3.38).

A.5.3.17 DIMENSION Statement

In Version 4.0, no restriction is placed on the number of array dimensions unless the **/4Ys** option is used in compiling, or the **$STRICT** metacommand is set (LR:5.3.12). In that case, arrays are restricted to seven dimensions.

Arrays in Version 3.3 are always restricted to seven dimensions (LR:3.2.10).

In Version 4.0, lower array-dimension bounds can be specified explicitly and can be positive, negative, or 0. If a lower dimension bound is not specified, it is 1 by default.

The upper and lower bounds are checked according to the following rules:

- If the upper and lower dimension bounds are constants, the compiler verifies that the upper dimension bound is greater than or equal to the lower dimension bound. If it is not, the compiler generates an error message.
- If either the upper or the lower dimension bound is not a constant, the **/4Yb** compiler option must be used (or the **$DEBUG** metacommand must be in effect) if you want to verify that the upper bound is greater than or equal to the lower bound (LR:5.3.12).

Note

If all of an array's dimensions are declared with no lower bounds and with upper bounds of 1, no bounds checking is performed, even if **/4Yb** or **$DEBUG** is used. In this case, the array is treated the same as an adjustable-size array, except that the declared size of the array is used to determined whether or not huge addressing is used.

Dimension declarators in Version 3.3 do not include lower bounds; the lower bound is always 1 (LR:3.2.10).

In Version 4.0, a dimension declarator can be an arithmetic expression, unless the **/4Ys** option is used in compiling (or the **$STRICT** metacommand is specified). The result of the expression is truncated to an integer by an implicit use of the **INT** intrinsic function. If an arithmetic expression is used as a dimension declarator, it cannot contain function or array-element references. If a dimension declarator with variables is used to declare an adjustable-size array, the variables either must be formal arguments to a routine or must exist in a common block. Also, the array itself must be a formal argument (LR:5.3.12).

A.5.3.18 DO Statement

In Version 4.0, loop indices in a **DO** statement can be integer, real, or double-precision expressions. (The new formula for determining the loop iteration count is shown in Section 5.3.13 of the *Microsoft FORTRAN Compiler Language Reference.*)

In Version 3.3, loop indices in a **DO** statement must be integer expressions (LR:3.2.11).

A.5.3.19 INQUIRE Statement

In Version 4.0, the **INQUIRE** statement can include the **BINARY=** option to indicate whether the file (or the file connected to the unit) specified in the statement is in binary format. It can also include the **BLOCKSIZE=** option, which reports the I/O buffer size for the file (LR:5.3.32).

In Version 3.3., these options do not appear.

A.5.3.20 PAUSE Statement

In Version 4.0, the **PAUSE** statement allows the user to enter a blank line to return control to the program. It also allows the user to execute one or more DOS commands before returning control to the program (LR:5.3.40). If this feature is used, the subdirectory containing **COMMAND.COM** should be part of the user's search path. While the program is suspended, the user can enter either of the following:

- A DOS command. After the command is executed, control is automatically returned to the program.
- The word **COMMAND** (uppercase or lowercase). After entering **COMMAND**, the user can enter as many DOS commands as desired, then type **EXIT** (uppercase or lowercase) to return control to the program.

In Version 3.3, the **PAUSE** statement only allows the user to enter a blank line to return control to the program (LR:3.2.32).

A.5.3.21 READ and WRITE Statements

In Version 4.0, the **READ** and **WRITE** statements can include the **FMT=***formatspec* option, which can appear at any position in the I/O control list (LR:5.3.43, 5.3.52). However, the **READ** statement must include a unit specifier if the **FMT=***formatspec* option is used.

In Version 3.3, a format specifier must be the second argument in a formatted **READ** or **WRITE** statement (LR:3.2.34, 3.2.42).

In Version 4.0, the unit specifier can be omitted in a **READ** statement of the following form:

READ *formatspec*, *iolist*

In this form of the **READ** statement, the unit is assumed to be the keyboard (*) unit (LR:5.3.43).

In Version 3.3, a unit specifier must be the first argument to a **READ** statement (LR:3.2.34).

A.5.3.22 STOP Statement

In Version 4.0, if the *message* parameter in a **STOP** statement is an integer, the program displays this value on the screen and returns the least-significant byte of this value to the operating system. (This is a value between 0 and 255, inclusive.) If the *message* parameter is not an integer, the program displays this value on the screen and returns 0 to the operating system (LR:5.3.49).

In Version 3.3, if the *message* parameter in a **STOP** statement is an integer, the program displays the specified integer (LR:3.2.39).

A.5.3.23 Type Statements

Type statements in Version 4.0 can be used to initialize the values of variables. However, variables that appear in **COMMON** and **EQUIVALENCE** statements cannot be initialized in this way.

Also, length specifiers in type statements in Version 4.0 can appear either before or after dimension declarators.

A.5.3.24 Conditional Compilation

In Version 4.0, the **/4cc** option of the **FL** command (or the **$DEBUG:***string* metacommand) can be used to specify conditional compilation. If one or more letters follows the **/4cc** option (or **$DEBUG** metacommand), lines in the source file that have one of those letters in column 1 are compiled into the program. Lines beginning with other characters are treated as comments (LR:6.2.1).

A.5.4 New Language Features

Sections A.5.4.1 – A.5.4.9 discuss new features for Version 4.0 of Microsoft FORTRAN and the changes you may have to make to source programs to take advantage of these features.

A.5.4.1 INTEGER*1 and LOGICAL*1 Data Types

Version 4.0 supports two new data types: **INTEGER*1** and **LOGICAL*1**.

An **INTEGER*1** value occupies 1 byte and can be any number in the range −127 to 127, inclusive. In an arithmetic expression, **INTEGER*1** is the lowest-ranked operand. If an **INTEGER*1** value is converted to an **INTEGER*2** value, the **INTEGER*1** value is used as the least-significant part of the **INTEGER*2** value, and the most-significant part is filled with copies of the sign bit (that is, it is sign extended). A new intrinsic function, **INT1,** is provided to convert values to type **INTEGER*1**.

A **LOGICAL*1** value occupies 1 byte of storage. The value of this byte is either 0 (**.FALSE.**) or 1 (**.TRUE.**).

A.5.4.2 C Strings

The following new string escape sequences from the C language have been added for Version 4.0:

Sequence	Character
\"	Double quote
\x*hh*	Hexadecimal bit pattern (where *h* is between 0 and F, inclusive)
\a	Bell

See Section 2.4.6.1 in the *Microsoft FORTRAN Compiler Language Reference* for more information about C strings.

A.5.4.3 Concatenation Operator

Version 4.0 supports the use of the concatenation operator (**//**) in character expressions. See Section 2.7.2 in the *Microsoft FORTRAN Compiler Language Reference* for more information about this operator.

A.5.4.4 New Intrinsic Functions

New intrinsic functions that perform data-type conversion and bit manipulation have been added for Version 4.0.

Data-Type Conversion

The following list summarizes the new intrinsic functions that are used for data-type conversion:

Function	Operation
INT1	Converts arguments to type **INTEGER*1**
HFIX	Converts arguments to type **INTEGER*2**
JFIX	Converts arguments to type **INTEGER*4**

See Section 3.11.3.1 in the *Microsoft FORTRAN Compiler Language Reference* for more information about these functions.

Bit Manipulation

In Version 4.0, several new intrinsic functions can be used to perform bitwise operations on variables. The following list summarizes these new intrinsic functions:

Function	Operation
IOR	Inclusive or
ISHL	Logical shift
ISHFT	Logical shift
ISHA	Arithmetic shift
ISHC	Rotate
IEOR	Exclusive or
IAND	Logical product
NOT	Logical complement
IBCLR	Bit clear
IBSET	Bit set
IBCHNG	Bit change
BTEST	Bit test

All of these functions except **NOT** and **BTEST** accept two arguments of type **INTEGER**, **INTEGER*1**, **INTEGER*2**, or **INTEGER*4** and return a result of the same type. If two arguments with different **INTEGER** types are given, the larger of the two types is returned (provided that it is also a legal type).

NOT accepts one argument of one of these types and returns a result of the same type. **BTEST** accepts two arguments of one of these types and returns a **LOGICAL** result. All of these functions can be passed as actual arguments.

See Section 3.11.3.15 of the *Microsoft FORTRAN Compiler Language Reference* for more information about these functions.

A.5.4.5 New Time and Date Functions

New subroutines and functions that get and set the date and time have been added for Version 4.0. The following list summarizes these functions:

Function	Operation
GETDAT	Gets the system date
GETTIM	Gets the system time
SETDAT	Sets the system date
SETTIM	Sets the system time

See Appendix C, "Additional Procedures," of the *Microsoft FORTRAN Compiler Language Reference* for more information about these functions.

A.5.4.6 Z Edit Descriptor

The new **Z** repeatable edit descriptor allows you to specify hexadecimal editing in input/output lists. This edit descriptor has the form **Z***w*, which specifies a field that is *w* characters wide. Hexadecimal digits A – F are output in uppercase. See Section 4.8.2.2 of the *Microsoft FORTRAN Compiler Language Reference* for rules for the use of this edit descriptor.

A.5.4.7 ENTRY Statement

The **ENTRY** statement specifies an entry point for a subroutine or external function. See Section 5.3.20 of the *Microsoft FORTRAN Compiler Language Reference*.

A.5.4.8 PRINT Statement

The **PRINT** statement specifies output to the screen (unit *). See Section 5.3.41 of the *Microsoft FORTRAN Compiler Language Reference* for a description of this statement.

A.5.4.9 $[NO]DECLARE, $[NO]FREEFORM, and $[NO]TRUNCATE Metacommands

Six new metacommands have been added to Version 4.0 of Microsoft FORTRAN: **$DECLARE**, **$NODECLARE**, **$FREEFORM**, **$NOFREEFORM**, **$TRUNCATE**, and **$NOTRUNCATE**.

The **$DECLARE** metacommand causes the compiler to display warning messages for variables that are not declared in type statements. The **$NODECLARE** metacommand suppresses these warnings. The **$NODECLARE** metacommand is the default. Note that the **/4Yd** compiler option has the same effect as the **$DECLARE** metacommand, and the **/4Nd** compiler option has the same effect as the **$NODECLARE** metacommand. See Section 6.2.2 of the *Microsoft FORTRAN Compiler Language Reference* for more information about these metacommands.

The **$FREEFORM** metacommand tells the compiler that the source program ignores the standard FORTRAN column restrictions (labels in columns 1 – 5, continuation characters in column 6, statements in columns 7 – 72, and any columns beyond 72 ignored). The **$NOFREEFORM** metacommand tells the compiler that the source program observes these column restrictions. **$NOFREEFORM** is the default. Note that the **/4Yf** compiler option has the same effect as the **$FREEFORM** metacommand, and the **/4Nf** compiler option has the same effect as the **$NOFREEFORM** metacommand. See Section 6.2.5 of the *Microsoft FORTRAN Compiler Language Reference* for more information about free-form programs.

The **$TRUNCATE** metacommand tells the compiler to generate warning messages for any names longer than six characters. This option makes it easier to port your programs to other systems. The **$NOTRUNCATE** metacommand tells the compiler to treat the first 31 characters in a name as significant. **$TRUNCATE** is the default. Note that the **/4Yt** compiler option has the same effect as the **$TRUNCATE** metacommand, and the **/4Nt** compiler option has the same effect as the **$NOTRUNCATE** metacommand. See Section 6.2.17 of the *Microsoft FORTRAN Compiler Language Reference* for more information about these metacommands.

Appendix B
Using Exit Codes

B.1 Introduction 337

B.2 Exit Codes with MAKE 337

B.3 Exit Codes with DOS Batch Files 338

B.4 Exit Codes for Programs in the FORTRAN Compiler Package 338

B.4.1 FL Exit Codes 339

B.4.2 LINK Exit Codes 339

B.4.3 CodeView™ Exit Codes 339

B.4.4 LIB Exit Codes 339

B.4.5 MAKE Exit Codes 340

B.4.6 EXEPACK Exit Codes 340

B.4.7 EXEMOD Exit Codes 340

B.4.8 SETENV Exit Codes 340

B.4.9 ERROUT Exit Codes 340

B.5 Exit Codes from FORTRAN Programs 341

B.1 Introduction

Most of the programs in the Microsoft FORTRAN Compiler package return an exit code (sometimes called an "errorlevel" code) that can be used by DOS batch files or other programs such as **MAKE**. If the program finishes without errors, it returns a code of 0. The code returned varies if the program encounters an error. This appendix discusses several uses for exit codes, and lists the exit codes that can be returned by each program in the Microsoft FORTRAN Compiler package.

B.2 Exit Codes with MAKE

The Microsoft Program Maintenance Utility (**MAKE**) automatically stops execution if a program executed by one of the commands in the **MAKE** description file encounters an error. The exit code is displayed as part of the error message, unless a minus sign (−) precedes the command line in the **MAKE** file.

For example, assume the **MAKE** description file TEST contains the following lines:

```
TEST.OBJ :      TEST.FOR
      FL /c TEST.FOR
```

If the source code in TEST.FOR contains a program error (but not if it contains a warning error), you would see the following message the first time you use **MAKE** with the **MAKE** description file TEST:

```
make: FL /c TEST.FOR - error 2
```

This error message indicates that the command FL /c TEST.FOR in the **MAKE** description file returned exit code 2.

B.3 Exit Codes with DOS Batch Files

If you prefer to use DOS batch files instead of **MAKE** description files, you can test the code returned with the **IF ERRORLEVEL** command. The following sample batch file, called COMPILE.BAT, illustrates how to do this:

```
FL /c %1
IF NOT ERRORLEVEL 1 LINK %1;
IF NOT ERRORLEVEL 1 %1
```

You can execute this sample batch file with the following command:

```
COMPILE TEST.FOR
```

DOS then executes the first line of the batch file, substituting TEST.FOR for the parameter %1, as in the following command line:

```
FL /c TEST.FOR
```

It returns a code of 0 if the compilation is successful, or a higher code if the compiler encounters an error. In the second line, DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), DOS executes the following command:

```
LINK TEST;
```

LINK also returns a code, which will be tested by the third line.

B.4 Exit Codes for Programs in the FORTRAN Compiler Package

An exit code of 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. Some programs can return various codes indicating different kinds of errors, while other programs return only 1 to indicate that an error occurred. The exit codes for each program are listed in Sections B.4.1 – B.4.9.

B.4.1 FL Exit Codes

Code	Meaning
0	No fatal error
2	Program error
4	System-level error (such as out of disk space or compiler internal error)

B.4.2 LINK Exit Codes

Code	Meaning
0	No error
1	Any **LINK** fatal error

B.4.3 CodeView™ Exit Codes

The Microsoft CodeView debugger does not return exit codes. However, it does display codes returned by programs that are run within the debugger. For example, if you run an executable file named `TEST.EXE` within the debugger and the program encounters an error that returns 1, you will see the following line:

```
Program terminated normally (1)
```

B.4.4 LIB Exit Codes

Code	Meaning
0	No error
1	Any **LIB** fatal error

B.4.5 MAKE Exit Codes

Code	Meaning
0	No error
1	Any **MAKE** fatal error

If a program called by a command in the **MAKE** description file produces an error, the exit code will be displayed in the **MAKE** error message.

B.4.6 EXEPACK Exit Codes

Code	Meaning
0	No error
1	Any **EXEPACK** fatal error

B.4.7 EXEMOD Exit Codes

Code	Meaning
0	No error
1	Any **EXEMOD** fatal error

B.4.8 SETENV Exit Codes

Code	Meaning
0	No error
1	Any **SETENV** fatal error

B.4.9 ERROUT Exit Codes

Code	Meaning
0	No error
1	Any **ERROUT** fatal error

B.5 Exit Codes from FORTRAN Programs

Code	Meaning
0	No error; normal program termination.
Nonzero	An error occurred, or a **STOP** [*message*] statement was executed.

FORTRAN run-time error messages return nonzero exit codes. Some messages in the M6*xxx* and R6*xxx* classes return specifically documented exit codes; for example, message R6000, stack overflow, returns an exit code of 255. These messages are listed in Section E.4.

The **STOP** [*message*] statement returns an exit code of 0 if the *message* is missing or is a character constant. If *message* is an integer, the least-significant byte (between 0 and 255, inclusive) is returned as the exit code.

If no **STOP** statement is executed and no error occurs, the exit code is 0.

Appendix C
Microsoft FORTRAN Record and File Formats

C.1 Introduction 345
C.2 Record Structures 345
C.2.1 Formatted Sequential Files 345
C.2.2 Formatted Direct Files 347
C.2.3 Unformatted Sequential Files 348
C.2.4 Unformatted Direct File 350
C.2.5 Binary Sequential Files 351
C.2.6 Binary Direct Files 352
C.3 Specifying Binary File Format 353

C.1 Introduction

This appendix describes the record structure in files created by Microsoft FORTRAN. For each file type, a diagram of the record format and a sample program that creates a file of that type are shown. See Chapter 4, "The Input-Output System," in the *Microsoft FORTRAN Compiler Language Reference* for more information about accessing files in FORTRAN programs.

C.2 Record Structures

The structure of a Microsoft FORTRAN file depends on the format of the data within the file and the file-access mode. Data in a file can be in one of three formats:

1. Formatted
2. Unformatted
3. Binary

FORTRAN files can have one of two access modes:

1. Sequential
2. Direct

C.2.1 Formatted Sequential Files

A formatted sequential file is a sequence of formatted records. Records may vary in length and may be empty. Each record ends with ASCII carriage-return (CR) and line-feed (LF) characters (ASCII hexadecimal codes 0D and 0A, respectively), as shown in Figure C.1.

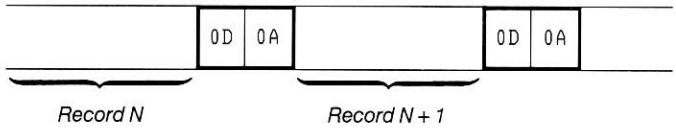


Figure C.1 Formatted Records in Formatted Sequential Files

■ Sample File Format and Program

Figure C.2 shows a sample formatted sequential file.

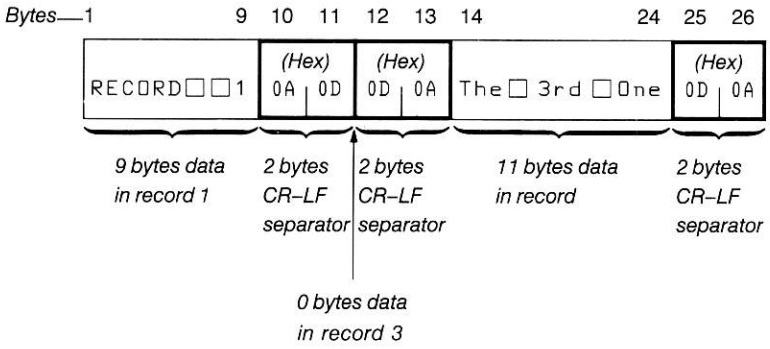


Figure C.2 Formatted Sequential File

The following program fragment creates the sample formatted sequential file shown in Figure C.2:

```
      I=4
      OPEN (33, FILE='FSEQ')
C (FORMATTED SEQUENTIAL BY DEFAULT)
      WRITE (33, '(A,I3)') 'RECORD', I/3
      WRITE (33, '()')
      WRITE (33, '(11HThe 3rd One)')
      CLOSE (33)
      END
```

C.2.2 Formatted Direct Files

A formatted direct file has basically the same structure as a formatted sequential file, except that all the records are exactly the same length. The record length is the same as the length specified in the **RECL=** specifier, plus 2 bytes that serve as record separators. If the record has been written, these bytes are ASCII carriage-return (CR) and line-feed (LF) characters (ASCII hexadecimal codes 0D and 0A, respectively). If the record has not been written, these bytes are undefined. Unwritten records contain undefined data. If data written to a record do not completely fill the record, they are padded with blanks out to the fixed record length.

■ Sample File Format and Program

Figure C.3 shows a sample formatted direct file.

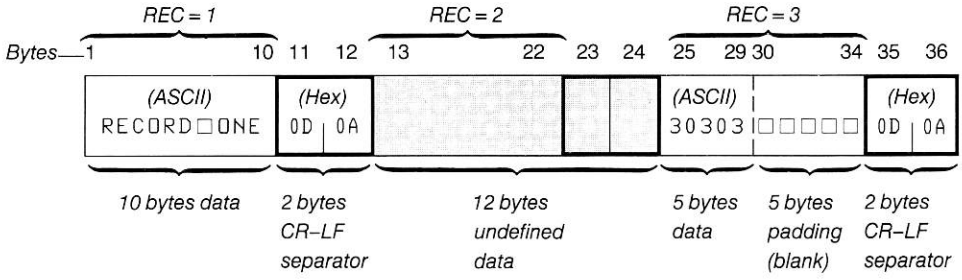


Figure C.3 Formatted Direct File

The following program fragment creates the sample formatted direct file shown in Figure C.3:

```
 OPEN (33, FILE='FDIR',FORM='FORMATTED',
+    ACCESS='DIRECT',RECL=10)
 WRITE (33, '(A)', REC=1) 'RECORD ONE'
 WRITE (33, '(I5)', REC=3) 30303
 CLOSE (33)
 END
```

C.2.3 Unformatted Sequential Files

An unformatted sequential file is a sequence of unformatted records. Records may vary in length. A logical record is represented as one or more physical blocks, each of which has the structure shown in Figure C.4.

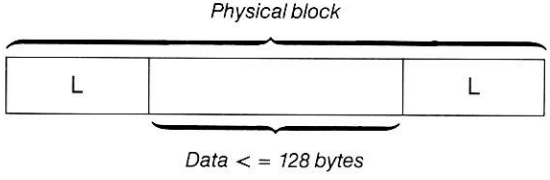


Figure C.4 Physical Block in Unformatted Sequential File

Each "L" in this illustration is a length byte that indicates the length of the data portion of the physical block. In the last physical block of the file, the following formula gives the number of bytes in the data portion:

MOD (*length* **128**)

Here *length* is the length of a logical record. In the physical block preceding the last physical block, the data portion contains 128 bytes, and the length byte contains 129. For example, if the size of a logical record is 140 bytes, the logical record has the format shown in Figure C.5.

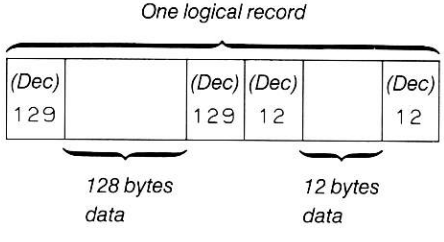
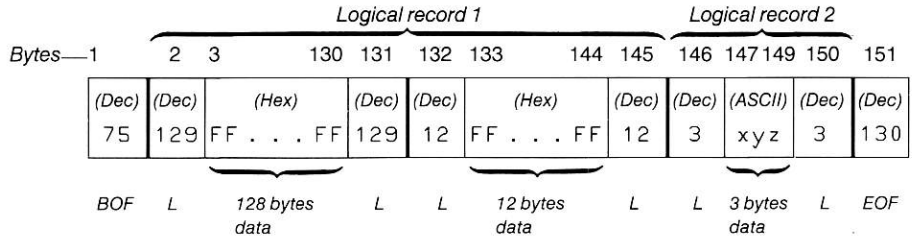


Figure C.5 Logical Record in Unformatted Sequential File

The first byte in the file is reserved and contains a value of 75. The last byte in the file is reserved and contains a value of 130. These bytes have no other significance. (Figure C.6 illustrates these bytes in a sample unformatted sequential file.)

■ Sample File Format and Program

Figure C.6 shows a sample unformatted sequential file.


Logical record 1
Logical record 2
Bytes—1 2 3 130 131 132 133 144 145 146 147 149 150 151
(Dec) (Dec) (Hex) (Dec) (Dec) (Hex) (Dec) (Dec) (ASCII) (Dec) (Dec)
75 129 FF . . . FF 129 12 FF . . . FF 12 3 xyz 3 130
BOF L 128 bytes data L L 12 bytes data L L 3 bytes data L EOF

L = Physical-block-length byte (0< =L< =129)
BOF = Beginning-of-file byte (75 decimal)
EOF = End-of-file byte (130 decimal)

Figure C.6 Unformatted Sequential File

The following program fragment creates the sample unformatted sequential file shown in Figure C.6:

```
      CHARACTER XYZ(3)
      INTEGER*4 IDATA (35)
      DATA IDATA /35 * -1/, XYZ /'x', 'y', 'z'/
C
C (-1 IS REPRESENTED BY FF FF FF FF HEXADECIMAL)
C
      OPEN (33, FILE='UFSEQ',FORM='UNFORMATTED')

C (SEQUENTIAL BY DEFAULT)
C
C WRITE OUT A 140-BYTE RECORD (ACTUAL DATA SIZE)
C FOLLOWED BY A 3-BYTE RECORD

      WRITE (33) IDATA
      WRITE (33) XYZ
      CLOSE (33)
      END
```

C.2.4 Unformatted Direct File

An unformatted direct file is a sequence of unformatted direct records. All records have the same length, which is the length given in the **REC=** specifier. No delimiting bytes separate records or otherwise indicate record structure.

Partial records can be written to an unformatted direct file. Version 4.0 of Microsoft FORTRAN pads these records up to the fixed record length with zeros (ASCII NUL characters); in files created by earlier versions of FORTRAN, random values may be used to pad records.

Unwritten records in the file contain undefined data.

■ Sample File Format and Program

Figure C.7 shows a sample unformatted direct file.

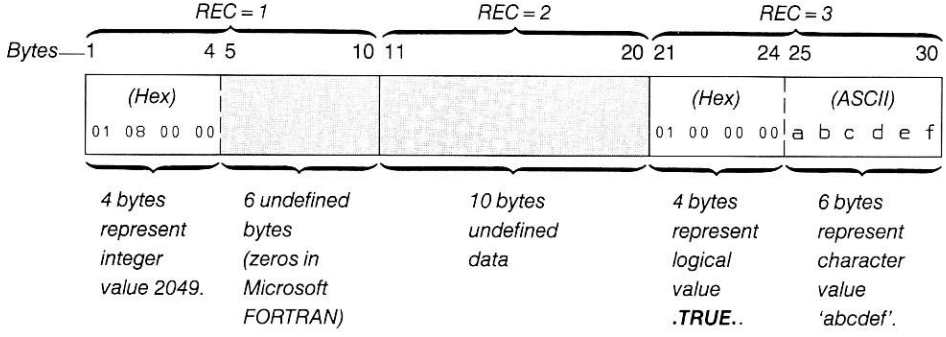


Figure C.7 Unformatted Direct File

The following program fragment creates the sample unformatted direct file shown in Figure C.7:

```
      OPEN (33, FILE='UFDIR', RECL=10,
     +      FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
      WRITE (33, REC=3) .TRUE., 'abcdef'
      WRITE (33, REC=1) 2049
      CLOSE (33)
      END
```

C.2.5 Binary Sequential Files

A binary sequential file is a sequence of values. No discernible record boundaries exist, and no special bytes indicate file structure. Data are read and written without changes in form or length. For any I/O list item, the sequence of bytes in memory is the sequence of bytes in the file.

■ Sample File Format and Program

Figure C.8 shows a sample binary sequential file.

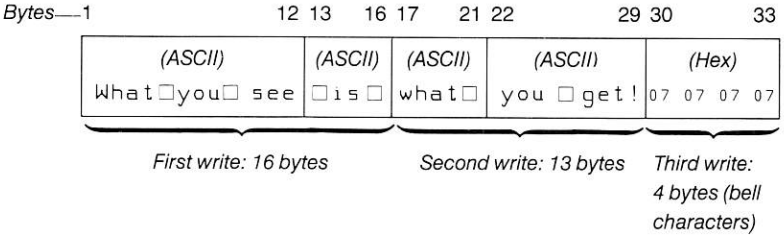


Figure C.8 Binary Sequential File

The following program creates the sample binary sequential file shown in Figure C.5:

```
$STORAGE:4
      INTEGER*1 BELLS(4)
      INTEGER IWYS(3)
      CHARACTER*4 CVAR
      DATA BELLS /4*7/
      DATA CVAR /' is '/,IWYS/'What',' you',' see'/

C THIS PROGRAM WRITES THE SENTENCE
C 'What you see is what you get!'
C FOLLOWED BY FOUR BELL CHARACTERS (07 HEXADECIMAL)

      OPEN (33, FILE='BSEQ',FORM='BINARY')

C (SEQUENTIAL BY DEFAULT)

      WRITE (33) IWYS, CVAR
      WRITE (33) 'what ', 'you get!'
      WRITE (33) BELLS
      CLOSE (33)
      END
```

C.2.6 Binary Direct Files

A binary direct file is identical in structure to an unformatted direct file, except for the following:

- Partial records are not padded with zero bytes; the unused portion of the record contains undefined data.
- A single read or write operation can be used to transfer more data than a record contains by continuing the operation into the next record(s) of the file. (Performing such an operation on an unformatted direct file would cause an error.) Valid I/O operations for unformatted direct files produce identical results when they are performed on binary direct files, provided the operations do not depend on zero padding in partial records.

■ Sample File Format and Program

Figure C.9 shows a sample binary direct file.

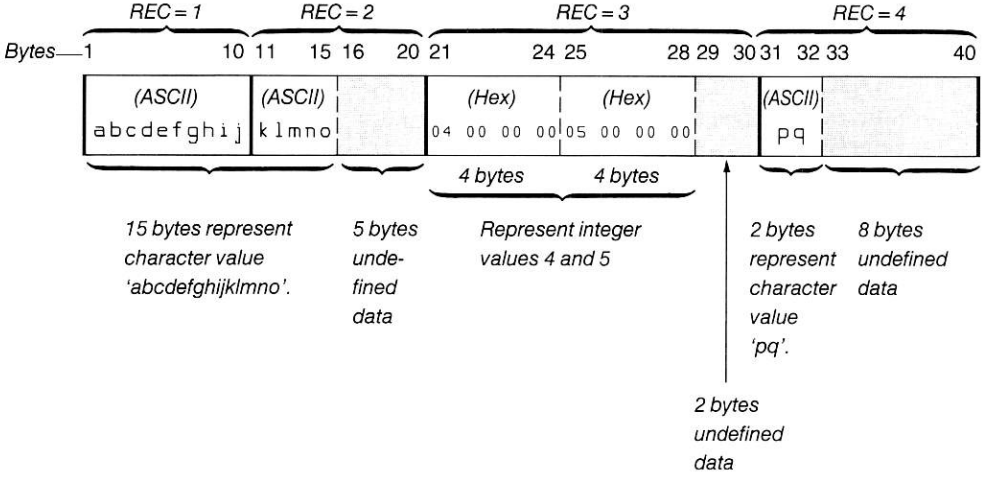


Figure C.9 Binary Direct File

The following program fragment creates the sample binary direct file shown in Figure C.9:

```
$STORAGE:4
      OPEN (33, FILE='BDIR',RECL=10,
     +FORM='BINARY',ACCESS='DIRECT')
      WRITE (33, REC=1) 'abcdefghijklmno'
      WRITE (33) 4,5
      WRITE (33,REC=40 'pq'
      CLOSE (33)
      END
```

C.3 Specifying Binary File Format

Specifying **FORM='BINARY'** is particularly useful when a FORTRAN program reads a file that was created by a program written in another language. Usually, such files do not have a particular internal structure or a file structure that corresponds to one of the standard FORTRAN file structures.

Appendix D
Handling 8087/80287 Floating-Point Exceptions

D.1 Introduction 357
D.2 Controlling the Processing Environment 358
D.2.1 The Status Word 359
D.2.2 The Control Word 360
D.3 Reading and Setting Status and Control Values 362
D.3.1 Store-Status-Word Procedure (SSWRQQ) 362
D.3.2 Store-Control-Word Procedure (SCWRQQ) 363
D.3.3 Load-Control-Word Procedure (LCWRQQ) 363

D.1 Introduction

The five exceptions to floating-point arithmetic required by the IEEE standard are supported by the 8087/80287 coprocessor and the real-math support routines. Exceptions that would result in a NAN ("Not a Number") error message when enabled are enabled by default. The others are disabled.

These exceptions are not affected by the **$DEBUG** metacommand; instead, they are controlled by a "status" word and a "control" word.

Table D.1 contains the five exceptions and their default and alternative actions:

Table D.1

Floating-Point Exceptions

Floating-Point Exception	Explanation	Default Action	Alternative Action
Invalid operation	Any operation that results in a NAN, such as the square root of −1 or 0*INF	Enabled; gives run-time error message M6101	Disabled; returns a NAN
Divide by zero	r/0.0	Enabled; gives run-time error message M6103	Disabled; returns a properly signed INF (infinity)
Overflow	Operation results in a number greater than maximum representable number.	Enabled; gives run-time error message M6104	Disabled; returns INF
Underflow	Operation results in a number smaller than smallest valid representable number.	Disabled; returns zero	Enabled; gives run-time error message M6105
Precision (or inexact)	Occurs whenever a result is subject to rounding error	Disabled; returns properly rounded result	Enabled; gives run-time error message M6106

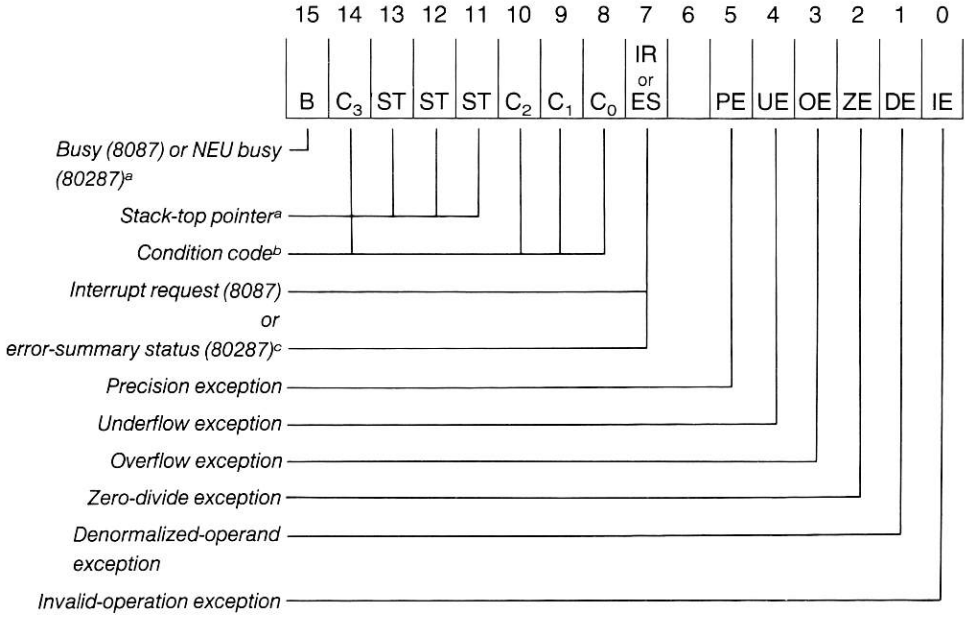
If any of these five exceptions are disabled, you will get either NAN, Infinite, or Indefinite values in your variables. If you print such a value, the output field will contain NAN, INF, or IND, padded with periods to the field width. If the output field has fewer than three spaces, only periods are printed.

D.2 Controlling the Processing Environment

Two memory locations can be used to control the 8086/80286 and the 8087/80287 processors: the status word and the control word. Sections D.2.1 and D.2.2 describe the uses of these memory locations.

D.2.1 The Status Word

Figure D.1 shows the format of the status word.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B C3 ST ST ST C2 C1 C0 IR or ES PE UE OE ZE DE IE
Busy (8087) or NEU busy (80287)a
Stack-top pointera
Condition codeb
Interrupt request (8087) or error-summary status (80287)c
Precision exception
Underflow exception
Overflow exception
Zero-divide exception
Denormalized-operand exception
Invalid-operation exception
(All other bits unused; may be either 1 or 0)

[a] The emulator ignores the settings of the stack-top-pointer and busy (or NEU-busy) bits.

[b] Refer to the Intel documentation for the interpretation of the condition-code bits.

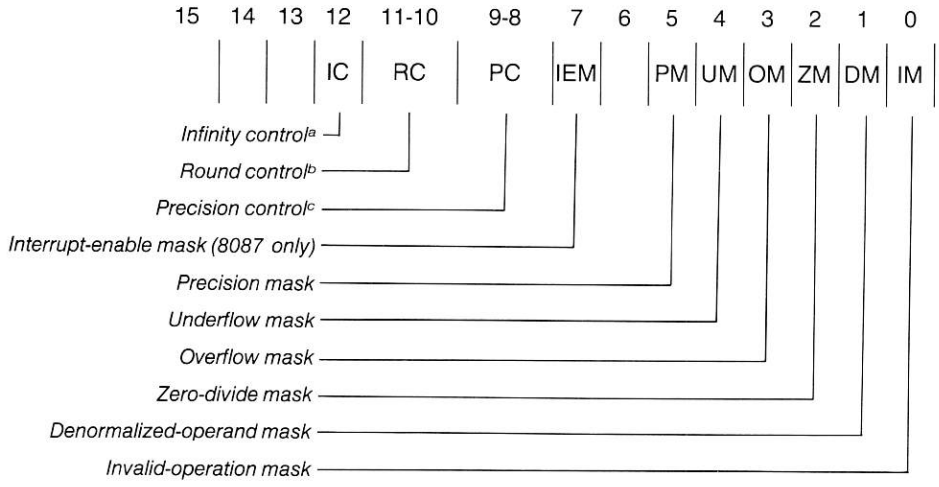
[c] On the 80287, ES is set if any unmasked exception bit is set; otherwise ES is cleared.

Figure D.1 Status-Word Format

When one of the exception conditions occurs, the appropriate bit in the status word is set. This flag remains set, indicating that the exception occurred, until the user clears it.

D.2.2 The Control Word

Figure D.2 shows the format of the control word.



(All other bits unused; may be either 1 or 0)

[a] *Infinity control*
- *0 = Projective*
- *1 = Affine*

[b] *Round control*
- *00 = Round nearest or even*
- *01 = Round down (toward – INF)*
- *10 = Round up (toward + INF)*
- *11 = Chop (truncate toward 0)*

[c] *Precision control*
- *00 = 24 bits of mantissa*
- *01 = (reserved)*
- *10 = 53 bits of mantissa*
- *11 = 64 bits of mantissa*

Figure D.2 Control-Word Format

When the bit corresponding to a given exception is set in the control word, that exception is masked, and the operation that caused the exception proceeds with a default action. If a bit corresponding to a given exception is reset, the corresponding exception generates an error message, halts the operation, and terminates the program. In either case, the exception is ORed into the status word. See Section 2.7.4 in the *Microsoft FORTRAN Compiler Language Reference* for more information on logical operators.

Besides masking exception conditions, the control word is used to set the following modes for the internal arithmetic required by the IEEE standard:

Mode	Description
Infinity Control	Chooses affine mode (the mode where the familiar +INF and −INF style of arithmetic is used) or projective mode (the mode where +INF and −INF are considered to be the same number). The principal effect of projective mode is to change the nature of comparisons: projective INF does not compare with anything but itself.
Round control	Rounds to nearest (or even), up, down, or chop.
Precision control	Determines the bit of the mantissa (24, 53, or 64) at which rounding should take place. Note that all results are stored to 64 bits regardless of the precision control. Precision control affects only the rounding in the internal form. When stored, any result is rounded to the storage precision again.

The current defaults for the control word are shown in the following list:

Mode or Mask	Default
Infinity control	Affine
Round control	Near
Precision control	64 bits
Interrupt-enable mask	Masked
Precision mask	Masked
Underflow mask	Masked
Overflow mask	Unmasked

Zero-divide mask	Unmasked
Denormalized-operand mask	Masked
Invalid-operation mask	Unmasked

Table D.2 defines the mask settings for the overflow, zero-divide, and invalid-operation exceptions associated with several optional control words. Control word 4914 specifies the default mask settings that are customary during 8087/80287 operations.

Table D.2

Mask Settings for Operation Exceptions

Control Word	Overflow	Zero Divide	Invalid-Operation
4914 = 16#1332	Unmasked	Unmasked	Unmasked
4915 = 16#1333	Unmasked	Unmasked	Masked
4918 = 16#1336	Unmasked	Masked	Unmasked
4919 = 16#1337	Unmasked	Masked	Masked
4922 = 16#133A	Masked	Unmasked	Unmasked
4923 = 16#133B	Masked	Unmasked	Masked
4926 = 16#133E	Masked	Masked	Unmasked
4927 = 16#133F	Masked	Masked	Masked

D.3 Reading and Setting Status and Control Values

This section describes the procedures and functions that you can use to read or set the values of the status and control words.

D.3.1 Store-Status-Word Procedure (SSWRQQ)

The **SSWRQQ** function returns the value of the status word. Use the following declaration for this procedure:

```
INTEGER*2 FUNCTION SCWRQQ
```

D.3.2 Store-Control-Word Procedure (SCWRQQ)

The **SCWRQQ** function returns the value of the control word. Use the following declaration for this procedure:

```
INTEGER*2 FUNCTION SCWRQQ
```

D.3.3 Load-Control-Word Procedure (LCWRQQ)

The **LCWRQQ** procedure sets the control word to a given value. **LCWRQQ** has the following declaration:

```
SUBROUTINE LCWRQQ(CW)
INTEGER*2 CW
```

The **INTEGER*2** argument passed to **LCWRQQ** is known as the "user's control word."

Always use **LCWRQQ** to change the control word. To ensure that special routines that handle stack exceptions and denormal propagation will work correctly, the control word and auxiliary variables must be set up in the way that **LCWRQQ** sets them up.

Important

Do not alter the 8087/80287 control word with an **FLDCW** instruction if you use an 8087/80287 coprocessor with a Microsoft language.

Since the denormal exception is not a part of the IEEE standard, **LCWRQQ** always alters the user's control word to mask denormals. The user cannot affect the handling of denormals with **LCWRQQ**.

Use the unmasked setting for the invalid-operation bit of the user's control word. The exception handler may ignore attempts to mask invalid exceptions, since it uses unmasked invalid exceptions to detect 8087/80287 stack overflow and underflow.

Error Messages

Appendix E
Error Messages and Limits

E.1 Introduction 367
E.2 Command-Line Error Messages 368
E.3 Compiler Error Messages 372
E.3.1 Compiler Fatal Error Messages 373
E.3.2 Compilation Error Messages 380
E.3.3 Recoverable Error Messages 426
E.3.4 Warning Error Messages 426
E.4 Run-Time Error Messages 434
E.4.1 Run-Time-Library Error Messages 435
E.4.2 Other Run-Time Error Messages 448
E.5 Linker Error Messages 454
E.6 LIB Error Messages 467
E.7 MAKE Error Messages 472
E.8 EXEPACK Error Messages 475
E.9 EXEMOD Error Messages 477
E.10 SETENV Error Messages 479
E.11 ERROUT Error Messages 480
E.12 Compiler and Linker Limits 481
E.12.1 Compiler Limits 481
E.12.1.1 Limits on Number of Names 483
E.12.1.2 Limits on Complicated Expressions 483
E.12.1.3 Limits on Character Expressions 484
E.12.2 Linker Limits 485

E.1 Introduction

This appendix lists error messages you may encounter as you develop a program, and describes actions you can take to correct the errors. The following list tells where to find error messages for the various components of Microsoft FORTRAN:

Component	Section
The command line used to invoke the Microsoft FORTRAN Compiler	Section E.2, "Command-Line Error Messages"
The Microsoft FORTRAN Compiler	Section E.3, "Compiler Error Messages"
The Microsoft FORTRAN run-time libraries and other run-time situations	Section E.4, "Run-Time Error Messages"
The Microsoft Overlay Linker (**LINK**)	Section E.5, "Linker Error Messages"
The Microsoft Library Manager (**LIB**)	Section E.6, "LIB Error Messages"
The Microsoft Program Maintenance Utility (**MAKE**)	Section E.7, "MAKE Error Messages"
The Microsoft EXE File Compression Utility (**EXEPACK**)	Section E.8, "EXEPACK Error Messages"
The Microsoft EXE File Header Utility (**EXEMOD**)	Section E.9, "EXEMOD Error Messages"
The Microsoft Environment Expansion Utility (**SETENV**)	Section E.10, "SETENV Error Messages"
The Microsoft STDERR Redirection Utility (**ERROUT**)	Section E.11, "ERROUT Error Messages"

Information on compiler and linker limits is found in Section E.12.

E.2 Command-Line Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

```
command line fatal error D1xxx: messagetext
command line error D2xxx: messagetext
command line warning D4xxx: messagetext
```

If possible, the compiler continues operation, printing error and warning messages. In some cases, command-line errors are fatal and the compiler terminates processing. The following messages indicate errors on the command line:

Number	Command-Line Error Message
D1000	UNKNOWN COMMAND LINE FATAL ERROR Contact Microsoft Technical Support An unknown error condition has been detected by the compiler. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
D1001	could not execute *'filename'* The specified file containing a compiler pass or the linker was found but for some reason could not be executed. An illegal **EXE** file format is the most likely cause.
D2000	UNKNOWN COMMAND LINE ERROR Contact Microsoft Technical Support An unknown error condition has been detected by the compiler. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
D2002	a previously defined model specification has been overridden Two different memory models were specified.

Number	Command-Line Error Message
D2003	missing source file name You must give the name of the source file to be compiled.
D2008	too many *option* flags, '*string*' Too many letters were given with a specific option (for example, with the **/O** option).
D2009	unknown option '*character*' in '*optionstring*' One of the letters in the given option was not recognized.
D2010	unknown floating point option The specified **/FP** option was not one of the valid options.
D2011	only one floating point model allowed Only one **/FP** option can be given on the command line. For example, the following **FL** command would cause this error: `FL /FPa test1.for /FPi test2.for`
D2012	too many linker flags on command line An attempt was made to pass more than 128 separate options and object files to the linker.
D2013	incomplete model specification The **/A***string* option requires all three characters (data-pointer size, code-pointer size, and segment setup) in *string*.
D2015	assembly files are not handled A file name with the extension **.ASM** was specified. The Microsoft FORTRAN Compiler cannot invoke **MASM** automatically, so it cannot assemble such files.
D2018	cannot open linker cmd file The response file used to pass object-file names and options to the linker could not be opened. This error can occur if another read-only file has the same name as the response file.

Number	Command-Line Error Message
D2019	`cannot overwrite the source file, '`*filename*`'` The source-file name was specified as an output-file name. The compiler does not allow compiler output files to overwrite the source file.
D2021	`invalid numerical argument '`*string*`'` An invalid number was read (for example, `/Gt -1`).
D2022	`cannot open help file, 'fl.hlp'` The **/HELP** compiler option was used, but the file containing the help messages (**FL.HLP**) was not in the current directory or in any of the directories specified by the **PATH** environment variable.

D4000	`UNKNOWN COMMAND LINE WARNING`
	`Contact Microsoft Technical Support`
	An unknown error condition has been detected by the compiler.
	Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.

D4001	`listing has precedence over assembly output` Two different listing options were chosen (for example, **/Fl** and **/Fa**). The assembly-language listing was not created.
D4002	`ignoring unknown flag '`*string*`'` One of the options given on the command line was not recognized and was ignored.
D4003	`80186/286 selected over 8086 for code generation` Both the **/G0** option and either the **/G1** or the **/G2** option were given; **/G1** or **/G2** takes precedence.

Number	Command-Line Error Message
D4004	`optimizing for time over space` Both the **/Os** option and the **/Ot** option were specified for optimizing. The **/Ot** option is used, since it takes precedence over **/Os**.
D4005	`could not execute '`*filename*`' ; please insert diskette and press any key.` The given file was not found in the current working directory or any of the other directories named in the **PATH** environment variable.
D4008	`non-standard model -- defaulting to large-model libraries` A nonstandard memory model was specified with the **/A***string* option. The library-search records in the object module are set to use the large-model libraries.
D4009	`threshold only for far/huge data, ignored` The **/Gt** option cannot be used in medium-model programs, which have near data pointers. It can be used only in large and huge models.
D4013	`combined listing has precedence over object listing` When **/Fc** is specified along with either **/Fl** or **/Fa**, the combined listing (**/Fc**) is created.
D4014	`invalid value` *number* `for '`*option*`'. Default` *number* `is used.` An out-of-range value was specified for an **FL** option. For example, the value for the **/Sp** or **/Sl** option may have been too large or too small.
D4017	`conflicting stack checking options - stack checking disabled` Conflicting stack-checking options (such as **/Ge** and **/Gs**) were given. As a result, stack checking was disabled.

E.3 Compiler Error Messages

The error messages produced by the Microsoft FORTRAN Compiler fall into the following four categories:

1. Fatal error messages
2. Compilation error messages
3. Recoverable error messages
4. Warning messages

The messages for each category are listed in Sections E.3.1 – E.3.4 in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the file name and line number where the error occurs. The following paragraphs discuss error-message format.

Fatal Error Messages

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename(*line*) : fatal error F1*xxx*: *messagetext*

After the compiler displays a fatal error message, it terminates without producing an object file or checking for further errors.

Compilation Error Messages

Compilation error messages identify actual program errors. These messages appear in the following format:

filename(*line*) : error F2*xxx*: *messagetext*

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Recoverable Error Messages

Recoverable error messages are informational only; they do not prevent compiling and linking. These messages appear in the following format:

filename(*line*) : error F3*xxx*: *messagetext*

Recoverable error messages are similar to warning error messages (see below), except that you cannot suppress them using the **/W** option. (See Section 3.3.9.3 for a description of this option.)

Warning Messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename(*line*) : warning F4*xxx* : *messagetext*

You can use the **/W** option to control the level of warnings that the compiler generates. See Section 3.3.9.3 for a description of this option.

E.3.1 Compiler Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it stops processing after printing the error message.

Number	Compiler Fatal Error Message
F1000	UNKNOWN FATAL ERROR Contact Microsoft Technical Support An unknown fatal error has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1001	Internal Compiler Error compiler file '*filename*' line '*number*' Contact Microsoft Technical Support An internal compiler error has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.

Number	Compiler Fatal Error Message
F1002	`out of heap space` The compiler ran out of dynamic memory space. This usually means that your program has many complex expressions. Try breaking expressions into smaller subexpressions.
F1003	`error count exceeds` *number*`; stopping compilation` The limit for compilation errors was exceeded.
F1005	`string too big for buffer` A string in a compiler intermediate file overflowed a buffer. This internal compiler error could result from initializing a very long character string with a **DATA** statement. Try decreasing the length of the character string.
F1006	`write error on compiler intermediate file` A write error occurred on a compiler intermediate file. This could be caused by a faulty disk.
F1008	*filename* `: cannot open include file` The specified include file could not be opened because it was not found in the source directory or in other directories specified by the include search paths given on the command line.
F1027	`DGROUP data allocation exceeds 64K` Allocation of variables to the default data segment exceeded 64K. For large- and huge-model programs, compile with the **/Gt** option to move items into separate segments.
F1031	`limit exceeded for nesting function calls` Function calls were nested to more than 30 levels.

Number	Compiler Fatal Error Message
F1032	cannot open object listing file *name* The compiler could not open the given object-listing file for writing. The file or disk may be write-protected, or the disk is full.
F1033	cannot open assembly language output file *name* The compiler could not open the given assembly-listing file for writing. The file or disk may be write-protected, or the disk is full.
F1035	expression too complex, please simplify The compiler could not generate code for a complicated expression. Try breaking the expression into simpler subexpressions and recompiling. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1036	cannot open source listing file *name* The compiler could not open the given source-listing file for writing. The file or disk may be write-protected, or the disk is full.
F1037	cannot open object file *name* The compiler could not open the given object file for writing. The file or disk may be write-protected, or the disk is full.
F1039	unrecoverable heap overflow in Pass 3 The compiler ran out of dynamic memory space. A subroutine may have too many symbols; simplify the subroutine and make it smaller.
F1041	cannot open compiler intermediate file - no more files The compiler was unable to open an intermediate file because too many files were already open.

Number	Compiler Fatal Error Message
F1043	`cannot open compiler intermediate file`

The compiler was unable to open an intermediate file. This could occur if the environment variable **TMP** was set to a nonexistent directory.

Try setting the environment variable **TMP** to an existing directory, or not setting **TMP** at all.

F1044 `out of disk space for compiler intermediate file`

The compiler ran out of disk space while writing to an intermediate file.

Try making more disk space available and recompiling.

F1045 `floating point overflow`

A compile-time evaluation of a floating or complex expression resulted in overflow, as shown in the following example:

```
real a,b,c
  a=10e30
  b=10e30
  c=a*b
```

F1050 *name* `: code segment too large`

The amount of object code in the named segment was larger than 64K.

F1051 `program too complex`

Your program caused the compiler to overflow one of its internal tables. For example, this error can occur if your program has too many labels.

Note that the **/4Yb** compiler option and the **$DEBUG** metacommand cause a large number of labels to be generated. If you encounter this message, try recompiling with the **/4Nb** option or changing the the **$DEBUG** metacommand to **$NODEBUG** in your source file and recompiling; or, if your file contains more than one procedure, try compiling the procedures in separate files.

Number	Compiler Fatal Error Message
F1900	`maximum memory-allocation size exceeded` The program tried to allocate more than approximately 1900 bytes at one time. This is the upper limit for the size of character constants. (See Section E.12, "Compiler and Linker Limits," for more information.)
F1901	`program too large for memory` The combination of heap space and stack space overflowed the memory configurations of the machine.
F1902	`statement stack underflow` This is an internal error. The compiler could not interpret the nesting of statements. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1903	`statement-nesting limit exceeded` Structured statements were nested too deeply. The maximum legal depth is about 40 statements and varies slightly depending on the type of statement. (See Section E.12, "Compiler and Linker Limits," for more information.)
F1904	`illegal command-line option` This error should never occur. If it does, please report it to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1907	`too many symbols` The program overflowed the internal symbol counter. There is no set upper limit on the number of symbols allowed in a source file. However, in any case, no more than 20,000 names are allowed in one module.

Number	Compiler Fatal Error Message
F1908	ASSIGN : too many format labels

The program overflowed the assigned format-label table.

This error probably occurred because an **INTEGER*1** variable, which has a limit of 127 labels, was specified. To avoid this error, use an **INTEGER*2** variable instead. (See Section E.12, "Compiler and Linker Limits," for more information.)

F1909 *filename* : include file nested too deeply

More than 10 include files were active at the same time. (See Section E.12, "Compiler and Linker Limits," for more information.)

F1910 *name* : unrecognized option

This is an internal error; the compiler driver, **FL**, caught an illegal option.

F1912 *filename* : cannot open file

The specified file could not be opened.

F1913 *name* : name too long

The specified internal-file name was more than 14 characters long. The compiler creates internal files in the directory specified by the **TMP** environment variable. If the combined length of the **TMP** environment variable and the unique internal name exceeds the name-buffer length, this message appears (see Section E.12.1, "Compiler Limits," for more information).

Create a smaller **TMP** environment variable. If no **TMP** variable is specified, this error should never occur.

F1914 cannot open internal files

Internal files could not be created.

F1917 unknown primitive type

An internal error has occurred.

Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

Number	Compiler Fatal Error Message
F1918	`missing symbol reference` An internal error has occurred. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1919	`unknown constant type` An internal error has occurred. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F1920	`illegal -A option` An invalid memory-model option was given for the **FL** command line.
F1921	`too many ENTRY statements` More than 32,000 **ENTRY** statements were used in this subprogram. (This error is not likely to occur; see Section E.12, "Compiler and Linker Limits," for more information.)
F1922	`integer string too long` An integer-constant string of digits overflowed an internal buffer. (This error should never occur in normal use.) Shorten the integer strings to legal value.
F1923	`CHARACTER constant too long` A constant of type **CHARACTER** can have a maximum of approximately 1900 characters. (See Section E.12, "Compiler and Linker Limits," for more information.)
F1924	`FORMAT string too long` A **FORMAT** statement can have a maximum of approximately 1900 characters. (See Section E.12, "Compiler and Linker Limits," for more information.)
F1925	`out of disk space for compiler internal file` The disk became full while the compiler was writing to an internal file.

Number	Compiler Fatal Error Message
F1926	`write error on compiler internal file` An error occurred while the compiler was writing to an internal file. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

E.3.2 Compilation Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

Number	Compiler Compilation Error Message

F2000	`UNKNOWN ERROR`
	`Contact Microsoft Technical Support`
	An unknown compilation error has occurred.
	Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F2001	`INTEGER value overflow` An **INTEGER** constant was too large to be of the specified type.
F2002	`Hollerith not allowed` Hollerith constants are not allowed when the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect).
F2003	`illegal base value` The specified base value was not between 2 and 36, inclusive.
F2004	`INTEGER constant must follow #` No alphanumerics within the legal range for the base immediately (allowing for white space) followed the number sign (**#**).

Number	Compiler Compilation Error Message
F2005	illegal REAL constant A floating-point constant was in an illegal form.
F2006	missing] following attribute string The closing bracket for an attribute list was missing.
F2007	opening quote missing The leading quote in a string value of an **ALIAS** attribute was missing.
F2008	unrecognized attribute An item used as an attribute was not a legal FORTRAN attribute.
F2009	*character* : illegal separator An attribute list did not end with a closing right bracket (]) and was not continued with a comma (,), or an illegal character was used in the list for the **$NOTLARGE** metacommand.
F2010	*name* : name too long; truncated The specified name was more than 31 characters long. (The limit on the length of names may be less in some environments; see Section E.12, "Compiler and Linker Limits," for more information.)
F2011	octal value too big for byte An octal value was not within the range 8#0 to 8#377.
F2012	*name* : already specified in $[NOT]LARGE The specified item appeared in the lists for both the **$LARGE** and the **$NOTLARGE** metacommands. (The message shows the metacommand that appears first in the source program.)
F2013	too many continuation lines Either the **/4Ys** compiler option was used in compiling (or the **$STRICT** metacommand was in effect), and more than 19 continuation lines were used.

Number	Compiler Compilation Error Message
F2015	`$DEBUG:'<debug list>' : string expected` A quoted string was expected after a **$DEBUG** metacommand.
F2017	`$INCLUDE:'<filename>' : string expected` A quoted string specifying a file name was expected after an **$INCLUDE** metacommand.
F2018	`$LINESIZE (or $PAGESIZE) : integer constant out of range` An integer constant less than 80 or greater than 132 was specified in a **$LINESIZE** metacommand, or a lower bound of less than 15 was specified in the **$PAGESIZE** metacommand.
F2019	`$LINESIZE (or $PAGESIZE) : integer constant expected` An integer constant was expected after a **$LINESIZE** or **$PAGESIZE** metacommand.
F2020	`$[NOT]LARGE already set` The **$LARGE** or **$NOTLARGE** metacommand appeared more than once in a procedure. (The metacommand that appeared more than once is indicated in the message.)
F2021	`$[NOT]LARGE illegal in executable statements` The **$LARGE** or **$NOTLARGE** metacommand appeared between subprograms or within the specification section of a subprogram. (The metacommand that appeared between subprograms or within the specification section is indicated in the message.)
F2022	`$MESSAGE:'<message>' : string expected` A quoted string containing a message was expected after a **$MESSAGE** metacommand.

Number	Compiler Compilation Error Message
F2023	`divide by 0` The second operand in a division operation (**/**) evaluated to 0, giving undefined results.
F2024	`mod by 0` The second operand in a **MOD** function evaluated to 0, giving undefined results.
F2027	`$STORAGE:<number> : 2 or 4 expected` A number other than 2 or 4 followed the **$STORAGE:** metacommand.
F2028	`$SUBTITLE:'<subtitle>' : string expected` A quoted string was expected following a **$SUBTITLE:** metacommand.
F2029	`$TITLE:'<title>' : string expected` A quoted string was expected following a **$TITLE:** metacommand.
F2030	`unrecognized metacommand` An unrecognized string followed the dollar sign (**$**) in the source file.
F2031	`closing quote missing` A quoted string did not end with a single quote (').
F2032	`zero-length CHARACTER constant` An illegal **CHARACTER** constant of length 0 was used in the program.
F2033	`Hollerith constant exceeds 1313 characters` A Hollerith constant exceeded the maximum legal length.
F2034	`zero-length Hollerith constant` An illegal Hollerith constant of length 0 was used in the program.

Number	Compiler Compilation Error Message
F2035	`Hollerith constant : text length disagrees with given length` A Hollerith constant was smaller than the size given in the length field of the Hollerith constant.
F2036	*character* `: non-FORTRAN character` A special character in the source file was not recognized.
F2037	`illegal label field` A nondigit value was used in a label field.
F2038	`zero-value label field` A label with the value 0 was used in the program. Labels must have values between 1 and 99,999, inclusive.
F2039	`free-form label too long` A label was more than five digits long.
F2040	`label on continuation line` A label was declared on a continuation line.
F2041	`first statement line must have ' ' or '0' in column 6` A continuation character was used on the first statement line in the program.
F2042	`label on blank line` A label was used on a line with no statements.
F2043	`alternate bases illegal` Alternate integer bases are not allowed if the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect).
F2101	`DO : too many expressions` A **DO** statement had more than three items following the equal sign (=).

Number	Compiler Compilation Error Message
F2102	I/O implied-DO list : list empty No items appeared in an I/O implied-**DO** list.
F2103	I/O implied-DO list : too many expressions More than three expressions appeared after the equal sign (=) in an I/O implied-**DO** list.
F2104	I/O implied-DO list : illegal assignment Only one assignment is legal in an I/O implied-**DO** list.
F2105	I/O implied-DO list : too few expressions Fewer than two expressions followed the equal sign (=) in an I/O implied-**DO** list.
F2106	I/O implied-DO list : assignment missing No assignment appeared in an I/O implied-**DO** list, or more than two expressions in the list were embedded in parentheses.
F2107	assignments in COMPLEX constant illegal An illegal embedded assignment appeared in a constant of type **COMPLEX**.
F2108	illegal assignment in parenthesized expression An illegal embedded assignment appeared in an expression enclosed in parentheses.
F2111	numeric constant expected A symbolic or numeric constant did not appear as part of a complex constant.
F2112	*name* : not symbolic constant The specified name was not a symbolic constant.
F2113	component of COMPLEX number not INTEGER or REAL A component of a **COMPLEX** number was not of type **INTEGER** or **REAL**.

Number	Compiler Compilation Error Message
F2114	`parser stack overflow, statement too complex` The statement being parsed was too large for the parser.
F2115	`syntax error` The source file contained a syntax error at the specified line.

F2124	`CODE GENERATION ERROR`
	`Contact Microsoft Technical Support`
	The compiler could not generate code for an expression. Usually this error occurs with a complicated expression.
	Try rearranging the expression. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

F2125	*name* `: allocation exceeds 64K` The specified item exceeded the limit of 64K. Huge arrays are the only items that are allowed to be larger than 64K.
F2126	*name* `: automatic allocation exceeds 32K` The subroutine or function *name* has an exceedingly large amount of compiler-generated temporary variables that take up more than 32,767 bytes. Try splitting the subroutine or function into smaller pieces.
F2127	`parameter allocation exceeds 32K` The storage space required for the arguments to a function exceeded the limit of 32K.
F2128	*name* `: huge array cannot be aligned to segment boundary` The specified array violated one of the restrictions imposed on huge arrays. See Section 9.4.1.2, "Huge Model," for more information on these restrictions.

Number	Compiler Compilation Error Message
F2200	*subprogram* : formal argument *name* : CHARACTER*(*) cannot pass by value Arguments that are passed by value must have a length that can be determined at run time. **CHARACTER*(*)** lengths are determined at run time.
F2201	*subprogram* : type redefined The type given in the specified **ENTRY**, **FUNCTION**, or **SUBROUTINE** statement was redefined. It was defined with a different type in an earlier subprogram.
F2202	*subprogram* : defined with different number of arguments The specified **ENTRY**, **FUNCTION**, or **SUBROUTINE** statement was defined or used earlier in the program with a different number of arguments.
F2203	*subprogram* : formal argument *name* : symbol-class mismatch The specified formal argument was defined previously with a different class. An **EXTERNAL** statement that passes a function to a variable, or a similar symbol-class mismatch, can cause this error.
F2204	*subprogram* : formal argument *name* : type mismatch The specified formal argument has a different type than was declared or used earlier in the program.
F2206	ENTRY seen before FUNCTION or SUBROUTINE An **ENTRY** statement appeared before any **FUNCTION** or **SUBROUTINE** statements in the program. An **ENTRY** statement can only appear in functions and subroutines.

Number	Compiler Compilation Error Message
F2207	ENTRY not in function or subroutine An **ENTRY** statement appeared in a subprogram that was not a function or subroutine. It may have appeared in the main program.
F2208	*name* : formal argument used as ENTRY The specified name was used as a formal argument in an earlier **ENTRY** statement or in the subprogram header in the current subprogram.
F2209	*name* : illegal as formal argument The symbol class of the formal argument was illegal. A formal argument can only be a variable, array, subroutine, function, or entry point.
F2210	*name* : formal argument redefined The specified formal argument appeared in the argument list more than once.
F2211	alternate RETURN only legal within subroutine An alternate **RETURN** statement was specified outside of a subroutine.
F2212	*subprogram* : subprogram used or declared before INTERFACE The specified subprogram was used or declared before the corresponding **INTERFACE** statement appeared in the program.
F2213	*subprogram* : already defined The specified subprogram was already defined in the current module.
F2214	*subprogram* : already used or declared with different symbol class The specified subprogram was used earlier in the program with a different class. For example, a subprogram that was used earlier in the program as a function and then declared as a subroutine would cause this error.

Number	Compiler Compilation Error Message
F2215	*subprogram* : ENTRY : CHARACTER lengths differ In a subprogram, if an entry name of type **CHARACTER** is used, then all the entry names in that subprogram must be of type **CHARACTER**. If one entry name is of type **CHARACTER*(*)**, then all must be of that type.
F2216	*subprogram* : CHARACTER and non-CHARACTER types mixed in ENTRY statements **CHARACTER** and non-**CHARACTER** types were mixed in a subprogram.
F2217	too many PROGRAM statements More than one **PROGRAM** statement appeared in the source file. Only one **PROGRAM** statement is allowed per program.
F2218	*name* : used or declared before ENTRY statement The name in an **ENTRY** statement was declared previously or was used previously in the same subprogram. This caused a symbol-class conflict that prevented the name from being used in an **ENTRY** statement.
F2219	*subprogram* : formal argument *name* : VALUE/REFERENCE mismatch An **INTERFACE** statement or prior call specified a different way of passing this argument than that specified in the current declaration.
F2220	*subprogram* : length redefined The length of a function was different when it was called than when it was defined.
F2221	*subprogram* : formal argument *name* : NEAR/FAR/HUGE mismatch The **NEAR**, **FAR**, or **HUGE** attributes were defined differently in the **INTERFACE** statement than in the subprogram definition or its arguments.

Number	Compiler Compilation Error Message
F2222	*name* : formal argument previously initialized The formal argument to an **ENTRY** statement appeared previously in a **DATA** statement within the same subprogram.
F2223	*subprogram* : formal argument *name* : subprogram passed by VALUE The formal argument had the **VALUE** attribute. Subprograms cannot be passed to items that have the **VALUE** attribute.
F2224	*name* : language attribute mismatch Language attributes were declared differently in the **INTERFACE** statement than in the subprogram declaration.
F2225	*name* : NEAR/FAR attribute mismatch The **NEAR** or **FAR** attribute was used differently in the **INTERFACE** statement than in the subprogram declaration.
F2226	*name* : VARYING attribute mismatch The **VARYING** attribute was not used in both the **INTERFACE** statement and the subprogram declaration.
F2227	*subprogram* : formal argument *name* : previously passed by value, now by reference A formal argument previously passed by value was passed by reference. The **VALUE** attribute should be specified for the formal argument.
F2228	*subprogram* : formal argument *name* : previously passed by reference, now by value A formal argument previously passed by reference was passed by value. The **REFERENCE** attribute should be specified for the formal argument.

Number	Compiler Compilation Error Message
F2229	*subprogram* : formal argument *name* : previously passed with NEAR, now with FAR or HUGE An address-length mismatch occurred. This is because an **INTERFACE** statement specifying the **FAR** or **HUGE** attribute for the formal argument was not given.
F2230	*subprogram* : formal argument *name* : previously passed with FAR or HUGE, now with NEAR An **INTERFACE** statement specifying the **NEAR** attribute for the formal argument was not given.
F2231	*name* : PROGRAM : name redefined The program name already exists as a global entity.
F2232	*subprogram* : formal argument *name* : Hollerith passed to CHARACTER formal argument Hollerith constants are allowed only with constants of types **INTEGER**, **REAL**, and **LOGICAL**.
F2233	*name* : previously called near A function that was previously declared or referenced with near addressing was used with a far call.
F2234	*name* : previously called far A function that was previously declared or referenced with far addressing was used with a near call.
F2301	*name* : EQUIVALENCE: formal argument illegal An item other than a local variable or array, or a variable or an array in a common block, appeared in an **EQUIVALENCE** statement.
F2302	*name* : EQUIVALENCE : not array In an **EQUIVALENCE** statement, an item that was not an array had an argument or subscript list attached to it.

Number	Compiler Compilation Error Message
F2303	*name* : EQUIVALENCE : array subscripts missing A construct such as x ( ) was used to declare the specified array. If no bounds are required, delete the parentheses.
F2304	*name* : EQUIVALENCE : nonconstant offset illegal A nonconstant offset was used for an array in an **EQUIVALENCE** statement.
F2305	*name* : nonconstant lower substring expression illegal The lower bound of a substring expression was not a constant in an **EQUIVALENCE** statement.
F2306	*name* : EQUIVALENCE : enclosing class too big Arithmetic overflow occurred while the offset of an array expression in an **EQUIVALENCE** statement was being calculated.
F2308	*name* : COMMON : length specification illegal It is illegal to specify the length of a type in a **COMMON** statement. Use a separate type statement to declare the length.
F2309	*name* : COMMON : attributes on items illegal The specified item in a **COMMON** statement had attributes attached to it. It is legal to declare only the common block itself with attributes.
F2310	*name* : COMMON (or EQUIVALENCE) : SUBROUTINE (or FUNCTION) name illegal A function or subroutine name was included in a **COMMON** or **EQUIVALENCE** statement. Only local variables and arrays are legal.

Number	Compiler Compilation Error Message
F2311	*name* : COMMON (or EQUIVALENCE) : preinitialization illegal Items in **COMMON** or **EQUIVALENCE** statements cannot be preinitialized in type-declaration statements. Use the standard notation for the **DATA** statement.
F2312	*name* : COMMON (or EQUIVALENCE) : formal argument illegal The specified formal argument was used in a **COMMON** or **EQUIVALENCE** statement.
F2313	*name* : COMMON (or EQUIVALENCE) : not an array or variable An item other than an array or variable was used in a **COMMON** or **EQUIVALENCE** statement.
F2314	*array* : COMMON : too big Arithmetic overflow occurred while the size of a common block was being calculated.
F2315	*array* : COMMON : array size nonconstant or zero A nonconstant or 0 value was used to dimension the array.
F2316	*name1* , *name2* : EQUIVALENCE : both in blank common block Two items specified in an **EQUIVALENCE** statement at different offsets were both in a blank common block. In the **EQUIVALENCE** statement, these items were specified to be at the same location in memory.
F2317	*name1* , *name2* : EQUIVALENCE : both in common block *commonblock* Two items specified in an **EQUIVALENCE** statement at different offsets were both in a named common block. These items were specified in the **EQUIVALENCE** statement to be at the same location in memory.

Number	Compiler Compilation Error Message
F2318	*name1* , *name2* : EQUIVALENCE : in different common blocks Two items in different common blocks were specified in an **EQUIVALENCE** statement.
F2319	*name* : EQUIVALENCE : extends blank common block forward In an **EQUIVALENCE** statement, it is illegal to increase the size of a blank common block by adding memory elements before the beginning common block declared in the **COMMON** statement.
F2320	*name* : EQUIVALENCE : extends common block *commonblock* forward In an **EQUIVALENCE** statement, it is illegal to increase the size of a named common block by adding memory elements before the beginning common block declared in the **COMMON** statement.
F2321	*name1* , *name2* : EQUIVALENCE : conflicting offsets Processing of an **EQUIVALENCE** statement detected two items that should have had the same offsets but did not. Inconsistent use of **EQUIVALENCE** statements caused this problem.
F2322	*name* : EQUIVALENCE : two different common blocks An **EQUIVALENCE** statement placed one item in two different common blocks.
F2323	*commonblock* : COMMON : size changed The size of the specified common block differed from the size allocated in a prior subprogram.
F2324	*commonblock* : COMMON : too big to be NEAR The specified common block, declared with the **NEAR** attribute, is larger than a segment.

Number	Compiler Compilation Error Message
F2325	*name* : COMMON : function or subroutine name The specified name was used as both a common-block name and a function or subroutine name.
F2326	*name* : already in COMMON The specified name appeared in a **COMMON** statement elsewhere in this subprogram.
F2327	*name* : EQUIVALENCE : needs at least two items An **EQUIVALENCE** statement had fewer than two items in a class.
F2328	*name* : already typed The specified item appeared in an earlier type statement in the same subprogram.
F2329	blank common cannot be HUGE In medium model, blank common items must be smaller than a single segment. Named common items do not have this restriction.
F2330	*name* : already dimensioned Array bounds appeared for the specified item in an earlier specification statement in the same subprogram.
F2331	*name* : types illegal on BLOCK DATA/COMMON/PROGRAM/SUBROUTINE The specified item was not one of the symbol classes that can be typed.
F2332	*name* : cannot initialize in type statements An attempt was made to initialize the specified item in a type statement while the **/4Ys** compiler option was used (or the **$STRICT** metacommand was in effect).

Number	Compiler Compilation Error Message
F2333	*name* : DIMENSION : not array The specified item in a **DIMENSION** statement (for example, an item already declared in an **EXTERNAL** or **PARAMETER** statement) was not an array.
F2334	*array* : already dimensioned The specified item had already been declared with bounds in a previous **COMMON**, **DIMENSION**, or type statement.
F2336	*array* : array bounds missing Both bound expressions were missing from the declaration of the specified array. At least an upper bound must be present.
F2337	*array* : * : not last array bound An assumed-size array was declared with an asterisk (*) that did not occur in the last bound.
F2338	*array* : bound size too small The bound size of the specified array was not a positive whole number. Bounds of adjustable-size arrays can be checked at run time. This compile-time error occurs only when the upper and (possibly implicit) lower bounds create a negative or zero element count for an array bound.
F2339	*array* : adjustable-size array not in subprogram The specified adjustable-size array was declared in a subprogram declared with a **PROGRAM** or **BLOCK DATA** statement. An adjustable-size array is legal only in an **ENTRY**, **FUNCTION**, or **SUBROUTINE** statement in a subprogram.

Number	Compiler Compilation Error Message
F2341	*letters* : IMPLICIT : only single letter allowed The upper or lower value of the range in an **IMPLICIT** statement was not a single character.
F2343	*letter1* , *letter2* : IMPLICIT : lower limit exceeds upper limit The upper letter in the range in an **IMPLICIT** statement had a smaller value than the letter in the lower range.
F2344	*letter* : already IMPLICIT The specified character already appeared in an **IMPLICIT** statement earlier in the same subprogram.
F2345	*name* : illegal use of SAVE (or EXTERNAL, INTRINSIC, PARAMETER) The specified *name* appeared earlier in a conflicting type statement.
F2346	*name* : INTRINSIC : unknown name The specified *name* is not the name of a supported intrinsic function.
F2348	*name* : already declared SAVE (or EXTERNAL, INTRINSIC, PARAMETER) The specified *name* was declared more than once with the same type of statement.
F2349	*name* : PARAMETER : nonconstant expression The specified item was declared with a nonconstant value in a **PARAMETER** statement.
F2351	*name* : repeated in formal-argument list The specified item was repeated in the formal-argument list to a statement function.

Number	Compiler Compilation Error Message
F2352	*name* : formal argument not local variable Only local variables can be used as formal arguments to statement functions.
F2354	*name* : statement function already declared The specified statement function was already declared in the current subprogram.
F2355	*name* : formal argument not a variable An argument list or substring operator for the specified item appeared in the formal-argument list to a statement function.
F2356	*name* : statement function : too few actual arguments More formal arguments than actual arguments were declared for a statement function.
F2357	*name* : statement function : too many actual arguments More actual arguments than formal arguments were declared for a statement function.
F2359	*type* : illegal length An illegal length specifier for the given type was used in a declaration. For example, REAL*13 would cause this error.
F2362	integer constant expression expected An integer value or integer constant expression was expected for an optional type-length specification.
F2363	length *value* : illegal type length A zero or negative length specifier was used in a type statement, or the length specifier was larger than the largest allowed for all types.

Number	Compiler Compilation Error Message
F2364	only C attribute legal on INTEGER type

An attribute other than the **C** attribute appeared with an **INTEGER** type statement.

F2365 attributes illegal on non-INTEGER types

Attributes in type statements are illegal, other than the **C** attribute on the **INTEGER** statement. Attributes were not put on the variables themselves.

F2366 DOUBLE PRECISION : length specifier illegal

A **DOUBLE PRECISION** statement included a length specifier. **DOUBLE PRECISION** is the same as **REAL*8**.

F2367 value *value* : INTEGER : range error

The specified constant was out of range for type conversion, or the type of an integer item was in conflict with the integer size specified in the **/4I** compiler option (or **$STORAGE** metacommand). For example, this error occurs for the following:

```
$STORAGE:2
         INTEGER*4 i
         i = 300000+30000
         i =      10* 4000
         i = -30000-30000
         END
```

To correct this error, use the appropriate **INT2** or **INT4** intrinsic function to make sure the appropriate (2- or 4-byte) arithmetic is performed on the variable.

F2368 *name* : truncated to 6 characters

When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), only six characters can appear in identifier names.

F2369 *name* : $ illegal in C name

A character in the specified name was illegal for a C variable. C variables allow only underscores (_) and alphanumeric characters in names.

Number	Compiler Compilation Error Message
F2370	length specification illegal When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), length specifications can only be used with **CHARACTER** type statements.
F2371	*name1* , *name2* : EQUIVALENCE : character and noncharacter items mixed Character and noncharacter items were mixed in an **EQUIVALENCE** statement.
F2372	*name* : more than 7 array bounds When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), an array cannot have more than seven bounds.
F2373	*name* : REFERENCE or VALUE only legal on formal arguments A **REFERENCE** or **VALUE** attribute was used with an item that was not declared in the formal-argument list for the routine. If the item is used in a type statement, then the attributed item must also appear in the formal-argument list of a subprogram. If the item appears in an **ENTRY** statement, include the attribute there instead.
F2374	*name* : attributes illegal on array bounds No attributes are allowed on items that are used when dimensioning arrays.
F2375	*name* : assumed-size array : cannot pass by value An assumed-size array was passed as an actual argument to a routine that had its formal argument declared with the **VALUE** attribute.
F2376	*name* : adjustable-size array : cannot pass by value An adjustable-size array was passed as an actual argument to a routine that had its formal argument declared with the **VALUE** attribute.

Number	Compiler Compilation Error Message
F2377	*name* : NEAR common block has HUGE item A common block declared with the **NEAR** attribute included item(s) that required the common block to be huge.
F2378	*name* : NEAR array bigger than segment An array declared with the **NEAR** attribute was larger than a segment.
F2379	*name* : item in common block crosses segment An item or an array element in a common block crossed a segment boundary. Items or arrays must be evenly aligned to signal boundaries when a common block crosses a segment.
F2380	*name* : VARYING illegal on symbol class The **VARYING** attribute was used on something other than a function or subroutine.
F2381	*commonblock* : NEAR/FAR/HUGE attribute mismatches default An attribute declared for the given common block was different from the attribute implicitly applied to the common block in an earlier subprogram. In medium-model programs, the **NEAR** attribute is used implicitly, unless the size of the common block requires the common block to be huge. In large-model programs, the **FAR** or **HUGE** attribute is used implicitly.
F2382	*commonblock* : *attribute* attribute mismatch with earlier NEAR/FAR/HUGE An attribute given in an earlier common-block declaration (possibly in a different subprogram) was different from the current attribute.

Number	Compiler Compilation Error Message
F2383	*name* : COMMON : character and noncharacter items mixed Character and noncharacter items cannot be mixed in a common block when the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect).
F2401	*name* : DATA : illegal address expression An illegal expression was used for the offset in a **DATA** statement. Only constant offsets are legal for items in **DATA** statements.
F2402	*name* : cannot initialize formal argument The item being initialized was a formal argument to a subprogram.
F2403	*name* : cannot initialize item in blank common block An attempt was made to use a **DATA** statement to initialize the specified item in a blank common block.
F2404	*name* : can only initialize common block in BLOCK DATA subprogram An attempt was made to initialize the specified item named in a common block in a **DATA** statement. Items in named common blocks can be initialized only in **BLOCK DATA** subprograms.
F2405	*name* : DATA : not an array or variable Only arrays and variables can be initialized in **DATA** statements.
F2406	*name* : repeat count not positive integer The repeat count for initialization of the specified item was not a positive integer value.

Number	Compiler Compilation Error Message
F2407	*name* : DATA : nonconstant item in initializer list A nonconstant value was used to initialize the specified item in a **DATA** statement.
F2408	*name* : DATA : too few constants to initialize item The **DATA** statement did not include enough constants to initialize the specified item.
F2409	*name* : nonstatic address illegal in initialization During processing of an implied-**DO** list in a **DATA** statement, the specified item did not have a static address. Make sure the item has constant subscript specifiers.
F2410	*name* : bound or increment not constant The specified item in implied-**DO** initialization in a **DATA** statement did not have constant bounds.
F2411	*name* : DATA : zero increment In a **DATA** statement, the increment value in the implied-**DO** list for the specified item must be set so the loop exits.
F2412	*name* : DATA : active implied-DO variable The specified implied-**DO** variable was used in nested **DATA** implied-**DO** initialization loops in a **DATA** statement.
F2413	*name* : DATA : implied-DO variable not INTEGER In **DATA** statements, only implied-**DO** variables of types **INTEGER*** *n* and **INTEGER** are legal.
F2414	*name* : DATA : not array-element name Only array elements can be initialized in implied-**DO** initializations in **DATA** statements.

Number	Compiler Compilation Error Message
F2415	DATA : too few constants to initialize names The constant list was exhausted before the initialization of the name list was complete.
F2416	*name*: bound or increment not INTEGER When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), only items of type **INTEGER** are allowed for **DATA** implied-**DO** loop bounds and increments. Otherwise, any arithmetic type is allowed and is truncated to type **INTEGER** by an implicit use of the **INT** intrinsic function.
F2417	DATA : iteration count not positive In the implied-**DO** list (...,*dovar*=*start*,*stop*,*inc*), if the increment *inc* is positive, then *start* must be greater than *stop*; if the increment *inc* is negative, then *stop* must be greater than *start*. If not, then the loop would execute zero times, which is not allowed.
F2500	*array* : adjustable-size array : used before definition An adjustable-size array was used before it was seen in an **ENTRY** statement.
F2502	*type* : cannot convert to *type* When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), constants cannot be converted between **CHARACTER** and non-**CHARACTER** types.
F2503	*intrinsic* : incorrect use of intrinsic function Invalid arguments were given for the specified intrinsic function.
F2504	*intrinsic* : multiple arguments The specified intrinsic function had more than one argument; only one is legal.

Number	Compiler Compilation Error Message
F2505	*intrinsic* : cannot convert FAR address to NEAR An item in the specified intrinsic function can only be referenced with addressing consistent with the **FAR** or **HUGE** attribute.
F2506	cannot convert type to *type* An invalid type conversion to **CHARACTER** or **LOGICAL** type was attempted.
F2508	*array*: array bound used array reference An expression having an array was used when declaring an adjustable-size array. Only simple variables in common blocks on the current subprogram's formal-argument list are allowed as variables in the bound expression.
F2510	*name* : symbolic constant : subscript illegal The specified symbolic constant had an array index or argument list.
F2511	*name* : symbolic constant : substring illegal The specified symbolic constant had a substring operator.
F2512	*name* : variable : argument list illegal The specified simple variable included an argument list.
F2513	*name* : not a variable The specified item was not a variable. A variable is expected in this context.
F2514	concatenation with CHARACTER*(*) A **CHARACTER*(*)** item was used in a concatenation operation. Only items with specified lengths are legal in concatenations.

Number	Compiler Compilation Error Message
F2515	`left side of assignment illegal` The left side of an assignment statement was illegal. Only variables, array elements, or function-return variables may appear on the left side of assignment statements.
F2516	*name* `: assignment using active DO variable illegal` An active **DO** variable was used in an assignment statement.
F2517	`illegal implied-DO list in expression` In this context, implied-**DO** statements are illegal in expressions.
F2519	*name* `: operation error with COMPLEX operands` A constant-folding error occurred. The number created would probably overflow the allowed storage. Use smaller numbers.
F2520	*name* `: operation error with REAL operands` A constant-folding error occurred. The number created would probably overflow the allowed storage. Use smaller numbers.
F2521	`negative exponent with zero base` A negative exponent was used with a zero-value base.
F2522	`division by zero` Division by zero occurred during constant folding.
F2523	`only comparisons by .EQ. and .NE. allowed for complex items` Only **.EQ.** and **.NE.** are legal as comparison operators for complex items.

Number	Compiler Compilation Error Message
F2524	non-numeric operand A nonarithmetic operand was specified with an arithmetic operator.
F2525	exponentiation of COMPLEX and DOUBLE PRECISION together illegal When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), exponentiation is illegal with bases of type **COMPLEX** having **DOUBLE PRECISION** exponents, or with bases of type **DOUBLE PRECISION** having **COMPLEX** exponents.
F2526	concatenation of expressions illegal An illegal concatenation operation occurred. If a noncharacter item is used in a concatenation, it must be a constant or it must be addressable.
F2527	noncharacter operand When the **/4Ys** compiler option is used (or when the **$STRICT** metacommand is in effect), concatenation operators can be used only with character operands.
F2528	nonlogical operand Logical operators (**.AND.**, **.OR.**, **.NOT.**, **.EQV.**, and **.NEQV.**) must be used with logical operands.
F2529	operands of relation not numeric or character Relational operators (**.LT.**, **.LE.**, **.GT.**, **.GE.**, **.EQ.**, and **.NE.**) must be used with arithmetic or character operands.
F2530	*name*: symbol class illegal here The class of the given symbol was illegal in this context.

Number	Compiler Compilation Error Message
F2531	*name*: bound not integer The **/4Ys** compiler option was used in compiling (or the **$STRICT** metacommand was in effect), and a substring had a noninteger substring-bound expression. If the **/4Ns** compiler option is used in compiling (or the **$NOTSTRICT** metacommand is in effect), any arithmetic expression is legal and is truncated to integers through an implicit use of the **INT** intrinsic function.
F2532	*name* : substring on noncharacter item An attempt was made to take the substring from an item that was not a character variable or array item.
F2533	*name* : lower substring bound exceeds upper bound The **/4Yb** compiler option was used (or the **$DEBUG** metacommand was in effect), and the value of the upper substring bound was less than the value of the lower substring bound.
F2534	*name* : upper substring bound exceeds string length The **/4Yb** compiler option was used (or the **$DEBUG** metacommand was in effect), and the upper substring bound was greater than the length of the item from which the substring was taken. This error occurs only if the length of the item was not specified (that is, if it was declared as a **CHARACTER*** *n* item).
F2535	*name* : lower substring bound not positive The **/4Yb** compiler option was used (or the **$DEBUG** metacommand was in effect), and the lower substring bound was less than or equal to 0. The minimum value for items of type **CHARACTER** is 1.

Number	Compiler Compilation Error Message
F2536	*array* : subscript *number* out of range The **/4Yb** compiler option was used (or the **$DEBUG** metacommand was in effect), and a local array or an array in a common block had a bound out of range.
F2537	*array* : array subscripts missing The specified array, which did not have array subscripts, was used in an expression.
F2538	*array* : subscript *number* : not integer When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), a subscripting expression used in the specified array must be of type **INTEGER**. Otherwise, it must be an arithmetic type that is truncated to **INTEGER** by an implicit use of the **INT** intrinsic function.
F2539	*array* : too few array subscripts Not enough subscripts were given when the specified array was used in an expression.
F2540	*array* : too many array subscripts Too many subscripts were given when the array was used in an expression.
F2541	cannot convert between CHARACTER and non-CHARACTER constants If the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), constants cannot be converted between **CHARACTER** and non-**CHARACTER** types.
F2542	one numeric, one character operand If the **/4Ys** compiler option is used in compiling (or the **$STRICT** metacommand is in effect), both operands used with relational operators must be character or both must be arithmetic. Operands of different types cannot be mixed.
F2559	*array* : array bound used illegal variable Only variables in common blocks or variables in the formal-argument list to the current subprogram are legal when declaring adjustable-size arrays.

Number	Compiler Compilation Error Message
F2560	*array* : array bound used intrinsic call Only variables in common blocks or variables in the formal-argument list to the current subprogram are legal when declaring adjustable-size arrays.
F2561	*array* : array bound used function call Only variables in common blocks or variables in the formal-argument list to the current subprogram are legal when declaring adjustable-size arrays.
F2562	cannot pass CHARACTER*(*) by value The program tried to pass by value an item of type **CHARACTER*(*)**. This is illegal because the length of such items is not known at compile time. Actual arguments with a length of *n* can be passed to **CHARACTER*** *n* items, and these items can be passed by value, if required.
F2563	incompatible types for formal and actual arguments The types of the formal and actual arguments did not match. Formal and actual arguments must have the same types (except for arguments of type **CHARACTER**, where the lengths can differ).
F2564	incompatible types in assignment The expressions on the left and right sides of an assignment statement were of different types. For example, a logical expression cannot be assigned to an integer variable.
F2565	*operation* : COMPLEX : type conversion error An attempt was made to convert values of one type to types that hold a smaller range of values.
F2566	*operation* : REAL : type conversion error An attempt was made to convert values of one type to types that hold a smaller range of values.

Number	Compiler Compilation Error Message
F2567	LEN : illegal expression Only constants, symbols, concatenations, intrinsic type casts, and strings are allowed in the **LEN** intrinsic function.
F2568	*name* : illegal bound type Only integer items are allowed as array bounds when the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect). Otherwise, arithmetic types are allowed and are converted through an implicit use of the **INT** intrinsic function.
F2569	*name* : Hollerith constant passed by value A Hollerith constant must be passed by reference to a logical, real, or integer formal argument.

F2570	consecutive arithmetic operators illegal
	
	Unary plus and minus cannot follow other arithmetic operators. For example,
	
	`I = I**-1`
	
	is illegal;
	
	`I = I**(-1)`
	
	must be used instead.

F2571	consecutive relational operators illegal The **.NOT.** operator cannot follow another **.NOT.** operator.
F2572	illegal use of Hollerith constant Hollerith constants are only allowed in assignments, **DATA** statements, and subprogram references.
F2601	*intrinsic* : intrinsic function illegal as actual argument This intrinsic function is illegal as an actual argument. (Some specific versions of the generic intrinsic functions can be passed as actual arguments; see Section 3.11.3 of the *Microsoft FORTRAN Compiler Language Reference* for more information.)

Number	Compiler Compilation Error Message
F2604	*subprogram* : function : argument list missing The specified function was missing an argument list. At least an empty argument list (( )) must be present in expressions.
F2605	*subprogram* : function : substring operator illegal A substring operator was used illegally with the specified routine name. Substring operators can only be used with arrays and variables.
F2606	*subprogram* : formal argument *name* : type mismatch The type of a formal argument was different from the type of the actual argument used in the subprogram call.
F2607	*subprogram* : formal argument *name* : length mismatch The length of a formal argument was different from the length of the actual argument used in the subprogram call.
F2608	*subprogram* : formal argument *name* : Hollerith illegal with CHARACTER Hollerith constants can only be used with items of type **INTEGER**, **LOGICAL**, and **REAL** in **DATA** statements and subprogram references.
F2609	*subprogram* : formal argument * : actual not alternate-return label Because the specified formal argument was an alternate-return label, the current argument must also be an alternate-return label.

Number	Compiler Compilation Error Message
F2610	*subprogram* : formal argument *name* : not alternate-return label Because the specified formal argument was not an alternate-return label, the current argument must not be an alternate-return label.
F2611	*subprogram* : formal argument *name* : actual not subprogram The formal argument used in a subprogram declaration was a subprogram, but the actual argument was not.
F2612	*subprogram* : NEAR formal argument *name* : actual has FAR or HUGE address It is illegal to pass an item that must be addressed with far or huge addressing to a formal argument that must be addressed with near addressing.
F2615	*name* : not function or subroutine The specified item was not a function or subroutine. Check the item's use or declaration earlier in the program.
F2616	*subprogram* : illegal use of function or subroutine The program tried to use a function as a subroutine or use a subroutine as a function.
F2617	*subprogram* : adjustable-size array *array* : cannot pass by value An attempt was made to pass an adjustable-size array by value.
F2618	*subprogram* : cannot use CHARACTER*(*) function **CHARACTER*(*)** functions cannot be directly referenced. They can only be passed as actual arguments.

Number	Compiler Compilation Error Message
F2619	*name* : value argument bigger than segment An argument with a **VALUE** attribute was too big to be passed onto the stack.
F2620	*subprogram* : formal argument *name* : subprogram mismatch The type of the formal argument to the subprogram was not the same as the actual argument. Both the formal and the actual argument must be subroutines or functions.
F2621	*name* : formal argument *name* : not subprogram The actual argument to the subprogram was a subprogram, but the formal argument was not a subprogram.
F2622	*name* : assumed-size array *array* : cannot pass by value An assumed-size array can only be passed by reference.
F2623	*name* : nonconstant CHARACTER length : cannot pass by value If a substring is used when passing a **CHARACTER*(*n*)** or **CHARACTER*(*)** argument to a formal argument declared with the **VALUE** attribute, then the lower and upper substring values must be constant. Otherwise, the length cannot be determined.
F2624	*subprogram* : too few actual arguments The number of actual and formal arguments for the given subprogram did not match. This practice is legal only when the **C** and **VARYING** attributes are specified for the subprogram.
F2625	*subprogram* : too many actual arguments The number of actual and formal arguments for the given subprogram did not match. This practice is legal only when the **C** and **VARYING** attributes are specified for the subprogram.

Number	Compiler Compilation Error Message
F2702	*iooption* : array subscript missing In this context, the array in the specified I/O option cannot appear without subscripts.
F2703	*iooption* : not *type* The specified I/O option required an item of a different type. For example, the **REC**=*rec* option requires an integer expression.
F2704	*iooption* : not a variable or array element The specified I/O option required a variable or an array element, as opposed to an arbitrary expression.
F2705	label *number* : not between 1 and 99999 Statement labels are restricted to the range 1 – 99,999; they must be one to five digits, not all of which are 0.
F2706	UNIT= * illegal for this statement The asterisk (*) unit specifier (console unit) cannot be specified for this I/O statement. Use the asterisk (*) unit specifier only with **READ**, **WRITE**, or **INQUIRE** statements. For **INQUIRE** statements, the asterisk (*) unit specifier is allowed only when the **/4Ns** compiler option is used (or the **$NOTSTRICT** metacommand is in effect).
F2707	illegal unit specifier The unit specifier in a **UNIT**= option was not an integer expression, asterisk (*), character variable, array element, array, or substring. A noncharacter array is a legal unit specifier if the **/4Ns** compiler option is used in compiling (or the **$NOTSTRICT** metacommand is in effect).
F2708	illegal format specifier The format specifier in a **FMT**= option was not a statement label, integer variable, character expression, character array, noncharacter array, or asterisk (*).

Number	Compiler Compilation Error Message
F2709	HUGE format illegal An array declared with a **HUGE** attribute that appeared in a **$LARGE** metacommand, or that spanned more than one segment, was used as a format specifier.
F2711	FAR format illegal in medium model Data allocated with the **FAR** attribute were used as a format specifier in a medium-model program.
F2712	*iooption* : appears twice The specified I/O option was used more than once in the same I/O statement.
F2714	I/O option *number* : <keyword=> missing The I/O option at position *number* in the option list appeared without a keyword. An I/O option without a keyword must not appear past the second position in the option list. Also, only **UNIT=** and **FMT=** options can appear without a keyword. If the **UNIT=** option appears without a keyword, it must be the first option in the option list. If the **FMT=** option appears without a keyword, it must follow a **UNIT=** option without a keyword. For example, OPEN (2, 'F.DOT') would produce the message I/O option 2: <keyword=> missing because the **FILE=** option is missing in the second option.
F2715	*iooption* : option illegal for this statement The given I/O option could not be used with this I/O statement.

Number	Compiler Compilation Error Message
F2716	INQUIRE : either UNIT= or FILE= needed The **INQUIRE** statement must have either a **UNIT=** option or a **FILE=** option, but not both.
F2717	UNIT= missing This I/O statement lacked a **UNIT=** option.
F2718	illegal I/O formatting for internal unit Internal units do not allow the use of unformatted or list-directed I/O. A format specifier other than asterisk (*) must be used.
F2719	REC= illegal on internal unit Direct-access I/O is illegal on internal units.
F2720	FORMAT : label missing A **FORMAT** statement lacked a statement label in the range 1–99,999.
F2721	no ASSIGN statements for FMT=<integer variable> The current format specifier had no corresponding **ASSIGN** statement to set the integer variable to a valid **FORMAT** statement label.
F2722	UNIT= : not between -32767 and 32767 An external unit number was out of range.
F2723	*iooption* : unrecognized value in option An invalid or misspelled value was used with the given I/O option. For example, ACCESS='DIREKT' and ACCESS='RANDOM' are both illegal.
F2724	RECL= required to open direct-access file When opening a file for direct access, the **RECL=** option is required.

Number	Compiler Compilation Error Message
F2725	`illegal input list item` An input list item was not a variable, array, array element, or substring.
F2726	*iooption*`: * illegal with this option` The asterisk (*) unit specifier (console unit) cannot be used with the given I/O control specifier.
F2727	*array* `: assumed-size array illegal here` An assumed-size array cannot be used in this context.
F2729	`FAR or HUGE I/O item illegal in medium model` Data items having the **FAR** or **HUGE** attribute cannot be used in I/O statements in medium-model programs.

F2730	*name* `: cannot modify active DO variable`
	A **DO** variable cannot be modified within its range. For example, the following program fragments cause this error:
	`DO 100 I = 1,10`
	`OPEN (33, IOSTAT = I)`
	`100 CONTINUE`
	
	`READ (*,*) (I, I = 1,10)`

F2731	*iooption* `: noncharacter array nonstandard` If the **/4Ys** compiler option is used in compiling (or the **$STRICT** metacommand is in effect), standard forms of the language must be used. In these cases, only character variables, arrays, array elements, and substrings are legal as I/O specifiers.
F2733	*iooption* `: option nonstandard` The specified I/O option is not part of standard FORTRAN 77; it cannot be given if the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect).

Number	Compiler Compilation Error Message
F2734	END= : illegal when REC= present In **READ** statements, the **REC=** and **END=** options cannot both be present.
F2735	REC= : illegal when FMT= * In **READ** and **WRITE** statements, the **REC=** option is illegal if list-directed I/O is in use.
F2736	LOCKING : nonstandard If the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), the **LOCKING** statement is prohibited.
F2737	*iooption* : lowercase in string nonstandard If the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), the value of the specified I/O option must be given in uppercase. For example, ACCESS='DIRECT' is legal in this case, but ACCESS='direct' is not.
F2738	*name* : HUGE internal units illegal An array used as an internal unit cannot be declared with the **HUGE** attribute or used in a **$LARGE** metacommand. The array cannot be larger than one segment.
F2739	*name* : record length too large for internal unit For a noncharacter array used as an internal unit, the element size multiplied by the element count (that is, the record length of the internal file) was too large.
F2740	RECL= : out of range The value of the **RECL=** option was less than or equal to 0 or exceeded the maximum legal value.
F2800	*name* : CHARACTER*(*) type illegal An item was declared with **CHARACTER*(*)** type, but it was not in the formal-argument list in the current subprogram.

Number	Compiler Compilation Error Message
F2801	no ASSIGN statements for assigned GOTO (or FMT=) The program unit had no **ASSIGN** statements for use with an assigned **GOTO** statement or assigned **FMT=** specifier.
F2803	*name* : ASSIGN : variable not INTEGER Only variables of type **INTEGER*** *n* or **INTEGER** are legal in **ASSIGN** statements.
F2804	*name* : ASSIGN : too many INTEGER*1 variables Only the first 127 **ASSIGN** statements may use variables of type **INTEGER*1** in a subprogram. This restriction is caused by the storage limitations of **INTEGER*1** items.
F2805	label *number* : redefined in program unit The specified label appeared earlier in the subprogram. Labels may not be defined more than once within a single subprogram unit. This error may also occur if a **DO** loop references a previously defined label.
F2806	DO-loop variable : not a variable A **DO**-loop variable was a symbolic constant, not an actual variable.
F2807	*name* : illegal use of active DO-loop variable It is illegal to use an active **DO**-loop variable as another **DO**-loop variable in a nested **DO** statement. Values cannot be assigned to **DO**-loop variables within **DO** loops.
F2808	DO-loop variable not INTEGER or REAL Only variables of type **INTEGER*** *n* and **REAL*** *n* are legal as **DO**-loop variables.
F2809	DO-loop expression not INTEGER or REAL Only expressions of type **INTEGER*** *n* and **REAL*** *n* are legal as **DO**-loop bounds.

Number	Compiler Compilation Error Message
F2810	`zero illegal as increment` Only nonzero increments are legal as **DO**-loop increments. (Otherwise, the loop would never exit.)
F2811	`IF or ELSEIF missing` No **IF** or **ELSEIF** statement matching an **ELSE** or **ELSEIF** statement appeared in the program.
F2812	`ENDIF missing` Not all **IF** – **ENDIF** blocks were exited before an **END** statement appeared.
F2813	`DO-loop label` *number* `: not seen` Not all **DO** loops were exited before an **END** statement appeared.
F2814	`IF, ELSEIF, or ELSE missing` No **IF**, **ELSEIF**, or **ELSE** statement matching an **ENDIF** statement appeared in the program.
F2815	`assigned GOTO variable not INTEGER` Only items of type **INTEGER*** *n* and **INTEGER** are legal for assigned **GOTO** variables.
F2816	`computed GOTO variable not INTEGER` Only items of type **INTEGER*** *n* and **INTEGER** are legal for computed **GOTO** variables.
F2817	`expression type not LOGICAL` Expression types for logical or block **IF** statements must be of type **LOGICAL**[[***** *n*]].
F2818	`expression type not INTEGER or REAL` Expression types for arithmetic **IF** statements must be of type **INTEGER**[[***** *n*]] or **REAL**[[***** *n*]].

Number	Compiler Compilation Error Message
F2819	illegal statement after logical IF Only single-line statements can follow logical **IF** statements. All executable statements except **DO**, **ELSE**, **ELSEIF**, **END**, **ENDIF**, block **IF**, and logical **IF** can follow a logical **IF** statement.
F2820	block label *number* : must not be referenced Labels that appear on **ELSE** and **ELSEIF** statements cannot be referenced.
F2821	label *number* : previously used as executable label The specified label, previously referenced as an executable label, was used as a label for a **FORMAT** statement or specification statement.
F2822	label *number* : previously used as FORMAT label The specified label, previously referenced as a label for a **FORMAT** statement, was used as an executable label or a label for a specification statement.
F2823	DO-loop label *number* : out of order The specified termination label for a **DO** statement was out of order. **DO**-loop labels may have been reversed.
F2824	assigned and unconditional GOTO illegal here Assigned and unconditional **GOTO** statements cannot terminate **DO** loops.
F2825	block and arithmetic IF illegal here Block and arithmetic **IF** statements cannot terminate a **DO** loop.
F2826	statement illegal as DO-loop termination An **ELSE**, **ELSEIF**, **END**, **ENDIF**, **FORMAT**, **RETURN**, or **STOP** statement cannot be used to terminate a **DO** loop.

Number	Compiler Compilation Error Message
F2827	STOP (or PAUSE) : maximum of 5 digits The **STOP** and **PAUSE** statements allow only numeric values between 0 and 99,999, inclusive.
F2828	ASSIGN target not an INTEGER variable Only variables of type **INTEGER** are allowed as targets in **ASSIGN** statements.
F2829	STOP (or PAUSE) : illegal expression Only integers or character constants are legal in **STOP** and **PAUSE** statements.
F2830	END missing An **END** statement did not appear as the last statement in the module.
F2831	label *number* : must not be referenced The specified label appeared on a specification or **DATA** statement.
F2832	statement illegal in INTERFACE Only specification statements are legal in **INTERFACE** statements.
F2833	RETURN : integer or character expression required If the **/4Ys** compiler option is used for compiling (or the **$STRICT** metacommand is in effect), only integer or character expressions can follow the **RETURN** statement.
F2834	*name* : alternate RETURN missing An alternate **RETURN** statement was given in the specified subprogram when none was given in the subprogram declaration.

Number	Compiler Compilation Error Message
F2835	`statement out of order or END missing` A specification statement was embedded in execution statements, another statement appeared out of the legal statement sequence, or an **END** statement did not terminate a previous subprogram.
F2836	`statement out of order` A statement appeared out of the legal order of statements in the program. For example, a specification statement may have appeared with execution statements.
F2837	`label` *number* `: undefined` The specified label, which was referenced in a subprogram, was not defined.
F2838	`statement illegal in BLOCK DATA` Only type-specification and **DATA** statements are legal in **BLOCK DATA** subprograms.
F2839	`only variables allowed in assigned GOTO statements` Only variables are allowed in assigned **GOTO** statements.
F2840	*name* `: assumed-size array : not reference argument` Assumed-size arrays must be passed by reference. They cannot be local entities to the subprogram.
F2841	*name* `: adjustable-size array : not reference argument` Adjustable-size arrays must be passed by reference. They cannot be local entities to a subprogram.
F2901	`-4I2 or -4I4 expected` Only 2- and 4-byte default integer and logical values are supported.

Number	Compiler Compilation Error Message
F2902	`-4Y and -4N : both options used for argument` The **$DO66** or **$FREEFORM** metacommand was specified in both the **/4Y** and the **/4N** compiler options.
F2993	`separator expected in format` When the **/4Ys** compiler option is used (or the **$STRICT** metacommand is in effect), a comma (,), colon (:), right parenthesis ()), or slash (/) is expected to separate items in a format except in the following cases: • Between a **P** edit descriptor and an immediately following **F**, **D**, **E**, or **G** edit descriptor • Before or after a slash (/) edit descriptor • Before or after a colon (:) edit descriptor
F2994	`\ or $ : nonstandard edit descriptor in format` The **** and **$** edit descriptors are not part of standard FORTRAN 77 but are extensions to the language. This error occurs only if the **/4Ys** compiler option is used (or if the **$STRICT** metacommand is in effect).
F2995	`Z : nonstandard edit descriptor in format` The **Z** edit descriptor is not part of standard FORTRAN 77 but is an extension to the language. This error occurs only if the **/4Ys** compiler option is used (or if the **$STRICT** metacommand is in effect).

E.3.3 Recoverable Error Messages

The messages listed below indicate potential problems but do not hinder compilation and linking. The **/W** compiler option has no effect on the output of these messages.

Number	Compiler Recoverable Error Message
F3606	*subprogram* : formal argument *name* : type mismatch The type of a formal argument was different from the type of the actual argument used in the subprogram call.
F3607	*subprogram* : formal argument *name* : length mismatch The length of a formal argument was different from the length of the actual argument used in the subprogram call.

E.3.4 Warning Error Messages

The messages listed below indicate potential problems but do not hinder compilation and linking.

Number	Compiler Warning Error Message
F4000	UNKNOWN WARNING Contact Microsoft Technical Support An unknown warning has occurred. Please report this condition to Microsoft Corporation using the Software Problem Report form at the back of this manual.
F4001	colon expected following ALIAS An **ALIAS** attribute had the wrong form. The correct form for **ALIAS** is the following: **ALIAS:***string*

Number	Compiler Warning Error Message
F4002	$DEBUG:'<debug-list>' illegal with $FREEFORM This form of the **$DEBUG** metacommand was used when the **$FREEFORM** metacommand was in effect.
F4003	$DECMATH not supported The **$DECMATH** metacommand is not supported in this version of FORTRAN.
F4006	metacommand already set A metacommand that may appear only once was reset.
F4007	metacommand must come before all FORTRAN statements This metacommand must appear before all FORTRAN statements.
F4008	characters following metacommand ignored Any characters that follow a fully processed metacommand are ignored.
F4010	*filename* : error closing file A system error occurred while the specified source file was being closed.
F4011	empty escape sequence A backslash (\) occurred at the end of a C string such as 'abc\'. It is replaced by a zero. The backslash should be removed.
F4014	*character* : nonalphabetic character in $DEBUG ignored A nonalphabetic character was included in the list for the **$DEBUG** metacommand. The characters a–z or A–Z are the only legal characters. Case is ignored.

Number	Compiler Warning Error Message
F4056	`overflow in constant arithmetic` The result of an operation exceeded #7FFFFFFF.
F4057	`overflow in constant multiplication` The result of an operation exceeded #7FFFFFFF.
F4058	`address of frame variable taken, DS != SS` The program was compiled with the default data segment (**DS**) not equal to the stack segment (**SS**), and the program tried to point to a frame variable with a near pointer.
F4059	`segment lost in conversion` The conversion of a far pointer (a full segmented address) to a near pointer (a segmented offset) resulted in the loss of the segmented address.
F4060	`conversion of long address to short address` The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segmented address.
F4061	`long/short mismatch in argument : conversion supplied` Actual and formal arguments of a function differed in base type. The type of the actual argument was converted to the type of the formal argument.
F4062	`near/far mismatch in argument : conversion supplied` Actual and formal arguments of a function differed in pointer size. The size of the actual argument was converted to the size of the formal argument.
F4063	*name* `: function too large for post-optimizer` The compiler tried to optimize a function but ran out of memory while doing so. It flagged the warning, skipped the optimization, and continued the compilation. To avoid this problem, break the functions in the program into smaller functions.

Number	Compiler Warning Error Message
F4064	procedure too large, skipping *optimization* optimization and continuing The compiler tried to perform the given type of optimization on a function but ran out of memory while doing so. It flagged the warning, skipped the given part of the optimization, and continued the compilation. To avoid this problem, break the function into smaller functions.
F4065	recoverable heap overflow in post-optimizer - some optimizations may be missed The compiler tried to optimize a function but ran out of memory while doing so. It flagged the warning, skipped the optimization, and continued the compilation. To avoid this problem, break the function into smaller functions.
F4066	local symbol table overflow - some local symbols may be missing in listings The compiler ran out of memory when it tried to collect the local symbols for source listings. Not all of the symbols are listed.
F4201	formal argument *name* : ENTRY : formal argument *name* : *attribute* : mismatch **VALUE** and **REFERENCE** attributes were mismatched in the declaration and use of an **ENTRY** statement.
F4202	*subprogram* : formal argument *name* : never used If a formal argument is never referenced, the compiler must assume a variable was meant for this argument. In medium model, if a function is passed to the formal argument, the wrong amount of storage may be allocated. This message is suppressed by any previous compiler error message (F2*xxx*).

Number	Compiler Warning Error Message
F4303	*name* : language attributes illegal on formal arguments A language attribute (**C** or **PASCAL**) was specified for a formal argument to the current routine. It has no effect.
F4313	*name* : not previously declared While the **/4Yd** compiler option was used (or the **$DECLARE** metacommand was in effect), *name* was not declared in a type statement before it was used.
F4314	*intrinsic* : declared with wrong type The specified name was declared with an incorrect type in an **INTRINSIC** statement. The incorrect type is ignored, and the correct type is used.
F4315	*name* : *attribute* illegal with attributes specified in same list The specified attribute contradicts an earlier attribute for the item in the same attribute list.
F4316	*name* : *attribute* illegal with attributes specified in earlier list The specified attribute contradicts an attribute in an earlier attribute list for the item.
F4317	*name* : *attribute* attribute repeated The specified attribute was already used once in an earlier attribute list for the item, and it should only have appeared in one attribute list.
F4318	*name* : *attribute* illegal on COMMON statements The specified attribute is illegal on common-block declarations.
F4319	*name* : *attribute* illegal on formal arguments The specified attribute cannot be used on formal arguments.

Number	Compiler Warning Error Message
F4320	*name* : *attribute* illegal on ENTRY statements The specified attribute cannot be used on **ENTRY** statements.
F4321	*name* : *attribute* illegal on subprogram statements The specified attribute cannot be used on **SUBPROGRAM** statements.
F4322	*name* : *attribute* illegal on variable declarations The specified attribute cannot be used on variable declarations.
F4323	*name* : *attribute* illegal on type declarations The specified attribute cannot be used on type declarations.
F4324	*attribute* : attribute repeated The specified attribute was repeated in the same attribute list.
F4326	*name* : EQUIVALENCE : nonconstant upper substring expression ignored The upper substring expression in an **EQUIVALENCE** statement was not a constant. Since the expression is not used in the addressing expression, it is ignored.
F4327	*name* : INTERFACE : not formal argument A variable was declared that was not given in the formal-argument list to the subprogram specified in the **INTERFACE** statement.
F4400	DATA : more constants than names Extra constants appearing in a constant list of a **DATA** statement were ignored.

Number	Compiler Warning Error Message
F4501	*array* : subscript *number* out of range The **/4Yb** compiler option was used in compiling (or the **$DEBUG** metacommand was in effect), and an array passed as an argument had a bound out of range. (This practice is legal for formal arguments because it is common in FORTRAN to declare the last bound to be 1.)
F4602	*name* : alternate RETURN statement missing The subprogram declaration where the specified name appeared had no alternate **RETURN** statement.
F4605	*name* : FAR formal argument *name* : passed HUGE array An array declared with a **HUGE** attribute was passed to a formal argument declared with a **FAR** attribute.
F4801	label *number* : used across blocks An executable statement label was referenced across a statement block. This situation may arise in the following cases: • When a **GOTO** statement uses a statement label in a different arm of an **IF – ELSE – ENDIF** statement • When the program jumps into a **DO** loop
F4802	no assigned GOTO or FMT= for ASSIGN statement An **ASSIGN** statement was used to assign a label to a variable in the subprogram, but the variable was not used.
F4803	*name* : FUNCTION : return variable not set A return variable specified in a **FUNCTION** statement was not set at least once in the function.
F4901	-4Y and -4N : both options used; -4Y assumed The **$DEBUG**, **$DECLARE**, **$LIST**, **$STRICT**, or **$TRUNCATE** metacommand was specified with both the **/4Y** and **/4N** compiler options. For example, **$DEBUG** was specified using both a **/4Yb** option and a **/4Nb** option.

Number	Compiler Warning Error Message
F4902	-W*number* : illegal warning level ignored This is an internal check. Microsoft FORTRAN supports only warning levels 0 and 1.
F4980	integer expected in format An edit descriptor lacked a required integer value.
F4981	initial left parenthesis expected in format A format did not start with a left parenthesis ().
F4982	positive integer expected in format An unexpected negative or 0 value was used in a format. Negative integer values can appear only with the **P** edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.
F4983	repeat count on nonrepeatable descriptor One or more **BN**, **BZ**, **S**, **SP**, **SS**, **T**, **TL**, **TR**, **/**, ****, **$**, **:**, or apostrophe (') edit descriptors had repeat counts associated with them.
F4984	integer expected preceding H, X, or P edit descriptor An integer did not precede a (nonrepeatable) **H**, **X**, or **P** edit descriptor. The correct formats for these edit descriptors are *n***H**, *n***X**, and *k***P**, respectively, where *n* is a positive integer and *k* is an optionally signed integer.
F4985	N or Z expected after B in format An illegal edit descriptor beginning with "B" was used. The only valid edit descriptors beginning with "B" are **BN** and **BZ**, used to specify the interpretation of blanks as nulls or zeros, respectively.

Number	Compiler Warning Error Message
F4986	`format nesting limit exceeded` More than 16 sets of parentheses were nested inside the main level of parentheses in a format.
F4987	`'.' : expected in format` A period did not appear between the *w* and *d* fields of a **D**, **E**, **F**, or **G** edit descriptor.
F4988	`unexpected end of format` An incomplete format was used. Improperly matched parentheses, an unfinished Hollerith (**H**) descriptor, or another incomplete descriptor specification can cause this error.
F4989	`'`*character*`' : unexpected character in format` A character that cannot be interpreted as a valid edit descriptor was used in a format.
F4990	`M field exceeds W field in I edit descriptor` The length of the *m* field specified in an **I** edit descriptor exceeded the length of the *w* field.

E.4 Run-Time Error Messages

Run-time error messages fall into two categories:

1. Error messages generated by the run-time library to notify you of serious errors. These messages are listed and described in Section E.4.1.
2. Floating-point exceptions generated by the 8087/80287 hardware or the emulator. These exceptions are listed and described in Section E.4.2, "Other Run-Time Error Messages."

E.4.1 Run-Time-Library Error Messages

The following messages may appear at run time when your program has serious errors. Run-time error-message numbers range from F6000 to F6999.

A run-time error message takes the following general form:

[[*sourcefile* (*line*) :]] run-time error F6*nnn* : *operation*[[(*filename*)]]
- *messagetext*

The *sourcefile* (*line*) information appears only when the **$DEBUG** metacommand is in effect.

For *operation*, one of the following may appear: BACKSPACE, CLOSE, ENDFILE, INQUIRE, LOCKING, OPEN, READ, REWIND, WRITE, or $DEBUG.

The *filename* of the file affected by *operation* is shown except when *operation* is $DEBUG.

The *messagetext* follows on the next line.

Number	Run-Time Error Message
F6096	array subscript expression out of range
	An expression used to index an array was smaller than the lower dimension bound or larger than the upper dimension bound. This message appears only if the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect).
F6097	CHARACTER substring expression out of range
	An expression used to index a character substring was illegal. See Section 2.4.6.2 of the *Microsoft FORTRAN Compiler Language Reference* for more information. This message appears only if the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect).
F6098	label not found in assigned GOTO list
	The label assigned to the integer-variable name was not specified in the label list of the assigned **GOTO** statement. This message appears only if the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect).

Number	Run-Time Error Message
F6099	INTEGER overflow
	This error occurs whenever integer arithmetic results in overflow, or when assignment to an integer is out of range. This message appears only if the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect).
F6100	INTEGER overflow on input
	An **INTEGER*n** item exceeded the legal size limits. An **INTEGER*1** item must be in the range −127 to 127. An **INTEGER*2** item must be in the range −32,767 to 32,767. An **INTEGER*4** item must be in the range −2,147,483,647 to 2,147,483,647.
F6101	invalid INTEGER
	Either an illegal character appeared as part of an integer, or a numeric character larger than the radix was used in an alternate radix specifier.
F6102	REAL indefinite (uninitialized or previous error)
	An illegal argument was specified for an intrinsic function (for example, SQRT(−1) or ASIN(2)). This error message does not always appear where the mistake was originally made. It may appear if the invalid value is used later in the program.
F6103	invalid REAL
	An illegal character appeared as part of a real number.
F6104	REAL math overflow
	A real value was too large. Floating-point overflows in either direct or emulated mode generate NAN (Not-A-Number) exceptions, which appear in the output field as asterisks (*) or the letters NAN.
F6200	formatted I/O not consistent with OPEN options
	The program tried to perform formatted I/O on a unit opened with **FORM='UNFORMATTED'** or **FORM='BINARY'**.

Number	Run-Time Error Message
F6201	list-directed I/O not consistent with OPEN options
	The program tried to perform list-directed I/O on a file that was not opened with **FORM='FORMATTED'** and **ACCESS='SEQUENTIAL'**.
F6202	terminal I/O not consistent with OPEN options
	The **ACCESS='SEQUENTIAL'** option and either the **FORM='FORMATTED'** or the **FORM='BINARY'** option were not included in the **OPEN** statement for a special device name such as **CON**, **LPT1**, or **PRN**. These options are required because special device names are connected to devices that do not support direct access. When a unit is connected to a terminal device, an **OPEN** statement that has the options **FORM='FORMATTED'** and **ACCESS='SEQUENTIAL'** results in carriage control. If the **FORM='BINARY'** and **ACCESS='SEQUENTIAL'** options appear in an **OPEN** statement, binary data transfer takes place.
F6203	direct I/O not consistent with OPEN options
	A **REC=** option was included in a statement that transferred data to a file that was opened with the **ACCESS='SEQUENTIAL'** option.
F6204	unformatted I/O not consistent with OPEN options
	If a file is opened with **FORM='FORMATTED'**, unformatted or binary data transfer is prohibited.
F6205	A edit descriptor expected for CHARACTER
	The **A** edit descriptor was not specified when a character data item was read or written using formatted I/O.
F6206	E, F, D, or G edit descriptor expected for REAL
	The **E**, **F**, **D**, or **G** edit descriptor was not specified when a real data item was read or written using formatted I/O.

Number	Run-Time Error Message
F6207	I edit descriptor expected for INTEGER

The **I** edit descriptor was not specified when an integer data item was read or written using formatted I/O.

F6208 L edit descriptor expected for LOGICAL

The **L** edit descriptor was not specified when a logical data item was read or written using formatted I/O.

F6209 file already open : parameter mismatch

An **OPEN** statement specified a connection between a unit and a file name that was already in effect. In this case, only the **BLANK=** option can have a different setting.

F6300 KEEP illegal for scratch file

STATUS='KEEP' was specified for a scratch file; this is illegal because scratch files are automatically deleted at program termination.

F6301 SCRATCH illegal for named file

STATUS='SCRATCH' should not be used in an **OPEN** statement that includes a file name.

F6302 multiple radix specifiers

More than one alternate radix for numeric I/O was specified.

F6303 illegal radix specifier

A radix specifier was not between 2 and 36, inclusive.

F6304 illegal STATUS value

An illegal value was used with the **STATUS=** option.

STATUS= accepts the following values:

- **'KEEP'** or **'DELETE'** when used with **CLOSE** statements
- **'OLD'**, **'NEW'**, **'SCRATCH'**, or **'UNKNOWN'** when used with **OPEN** statements

Number	Run-Time Error Message
F6305	`illegal MODE value` An illegal value was used with the **MODE=** option. **MODE=** accepts the values **'READ'**, **'WRITE'**, or **'READWRITE'**.
F6306	`illegal ACCESS value` An illegal value was used with the **ACCESS=** option. **ACCESS=** accepts the values **'SEQUENTIAL'** and **'DIRECT'**.
F6307	`illegal BLANK value` An illegal value was used with the **BLANK=** option. **BLANK=** accepts the values **'NULL'** and **'ZERO'**.
F6308	`illegal FORM value` An illegal value was used with the **FORM=** option. **FORM=** accepts the following values: **'FORMATTED'**, **'UNFORMATTED'**, and **'BINARY'**.
F6309	`illegal SHARE value` An illegal value was used with the **SHARE=** option. **SHARE=** accepts the values **'COMPAT'**, **'DENYRW'**, **'DENYWR'**, **'DENYRD'**, and **'DENYNONE'**.
F6310	`illegal LOCKMODE value` An illegal value was used with the **LOCKMODE=** option. **LOCKMODE=** accepts the values **'LOCK'**, **'NBLCK'**, **'NBRLCK'**, **'RLCK'**, and **'UNLCK'**.
F6311	`illegal record number` An invalid number was specified as the record number for a direct-access file. The first valid record number for direct-access files is 1.

Number	Run-Time Error Message
F6312	no unit number associated with *
	In an **INQUIRE** statement, the **NUMBER=** option was specified for the file associated with * (console).
F6313	illegal RECORDS value
	The **RECORDS=** option in a **LOCKING** statement specified a negative number.
F6314	illegal unit number
	An illegal unit number was specified.
	Legal unit numbers can range from −32,767 to 32,767, inclusive.
F6315	illegal RECL value
	A negative or zero record length was specified for a direct file.
	The smallest valid record length for direct files is 1.
F6400	BACKSPACE illegal on terminal device
	A **BACKSPACE** statement specified a unit connected to a terminal device such as a terminal or printer.
F6401	EOF illegal on terminal device
	An **EOF** intrinsic function specified a unit connected to a terminal device such as a terminal or printer.
F6402	ENDFILE illegal on terminal device
	An **ENDFILE** statement specified a unit connected to a terminal device such as a terminal or printer.
F6403	REWIND illegal on terminal device
	A **REWIND** statement specified a unit connected to a terminal device such as a terminal or printer.

Number	Run-Time Error Message
F6404	DELETE illegal for read-only file
	A **CLOSE** statement specified **STATUS='DELETE'** for a read-only file.
F6405	external I/O illegal beyond end of file
	The program tried to access a file after executing an **ENDFILE** statement or after it encountered the end-of-file record during a read operation.
	A **BACKSPACE**, **REWIND**, or **OPEN** statement must be used to reposition the file before execution of any I/O statement that transfers data.
F6406	truncation error : file closed
	This is a transient error. While the file was being truncated, it was temporarily closed.
	After a few minutes, the file should be run again. If this error message reappears, the file should be checked for characteristics, such as locking or permissions, that would prevent it from being accessed.
F6407	terminal buffer overflow
	More than 131 characters were input to a record of a unit connected to the terminal (keyboard). Note that the operating system may impose additional limits on the number of characters that can be input to the terminal in a single record.
F6408	comma delimiter disabled after left repositioning
	A comma could not be used as a field delimiter. This is because the use of commas as input field delimiters is disabled if left tabbing leaves the file positioned in a previous buffer.

Number	Run-Time Error Message
F6409	LOCKING illegal on sequential file

A **LOCKING** statement specified a unit that was not opened with **ACCESS='DIRECT'**.

F6410	file already locked or unlocked

The program tried to lock a file that was already locked or tried to unlock a file that was already unlocked.

F6411	file deadlocked

A **LOCKING** statement that included the **'LOCK'** or **'RLCK'** value tried to lock a file, but the file could not be locked after 10 attempts.

F6412	SHARE not installed

The **SHARE.COM** or **SHARE.EXE** file must be installed on your system before you can use the **LOCKING** statement, or the **SHARE=** option in an **OPEN** statement.

F6413	file already connected to a different unit

The program tried to connect an already connected file to a new unit.

A file can be connected to only one unit at a time.

F6414	access not allowed

This error is caused by one of the following occurrences:

- The file name specified in an **OPEN** statement was a directory.
- An **OPEN** statement tried to open a read-only file for writing.
- The file's sharing mode does not allow the specified operations (DOS Versions 3.0 and later only).

Number	Run-Time Error Message
F6415	file already exists
	An **OPEN** statement specified **STATUS='NEW'** for a file that already exists.
F6416	file not found
	An **OPEN** statement specified **STATUS='OLD'** for a file that does not exist.
F6417	too many open files
	The program exceeded the system limit on the number of open files allowed at one time. To fix this problem, change the **FILES=** command in the **CONFIG.SYS** file.
F6418	too many units connected
	The program exceeded the limit on the number of open files per program. Close any unnecessary files. See the **FILES=** command in the *Microsoft MS-DOS User's Guide* for more information.
F6419	illegal structure for unformatted file
	The file was opened with **FORM='UNFORMATTED'** and **ACCESS='SEQUENTIAL'**, but its internal physical-record structure was incorrect or inconsistent.
F6420	unknown unit number
	A statement such as **BACKSPACE** or **ENDFILE** specified a file that had not yet been opened. (The **READ** and **WRITE** statements do not cause this problem since, instead of generating this error, they prompt you for a file if the file has not been opened yet.)
F6421	file read-only or locked against writing
	The program tried to transfer data to a file that was opened in read-only mode or locked against writing.

Number	Run-Time Error Message
F6422	no space left on device
	The program tried to transfer data to a file residing on a device that was out of storage space.
F6500	file not open for reading or file locked
	The program tried to read from a file that was not opened for reading or was locked.
F6501	end of file encountered
	The program tried to read more data than the file contains.
F6502	positive integer expected in repeat field
	When the *r***c* form is used in list-directed input, the *r* must be a positive integer.
F6503	multiple repeat field
	In list-directed input of the form *r***c*, an extra repeat field was used. For example, `READ(*,*) I,J,K` with input `2*1*3` returns this error. The `2*1` means send two values, each 1; the `*3` is an error.
F6504	invalid number in list-directed input
	Some of the values in a list-directed input record were not numeric. The following example would cause this error: `123abc`
F6505	invalid string in list-directed input
	A string item was not enclosed in single quotation marks.
F6506	comma missing in list-directed COMPLEX input
	When using list-directed input, the real and imaginary components of a complex number were not separated by a comma.

Number	Run-Time Error Message
F6507	T or F expected in LOGICAL read

The wrong format was used for the input field for logical data.

The input field for logical data consists of optional blanks, followed by an optional decimal point, followed by a **T** for true or **F** for false. The **T** or **F** may be followed by additional characters in the field, so that **.TRUE.** and **.FALSE.** are acceptable input forms.

F6508 too many bytes read from unformatted record

The program tried to read more data from an unformatted file than the current record contained. If the program was reading from an unformatted direct file, it tried to read more than the fixed record length as specified by the **RECL=** option. If the program was reading from an unformatted sequential file, it tried to read more data than was written to the record.

F6509 H or apostrophe edit descriptor illegal on input

Hollerith or apostrophe edit descriptors were encountered in a format used by a **READ** statement.

F6510 illegal character in hexadecimal input

The input field contained a character that was not hexadecimal.

Legal hexadecimal characters are 0 – 9 and A – F.

F6600 internal file overflow

The program either overflowed an internal-file record or tried to write to a record beyond the end of an internal file.

F6601 direct record overflow

The program tried to write more than the number of bytes specified in the **RECL=** option to an individual record of a direct-access file.

Number	Run-Time Error Message
F6602	`list-directed number bigger than record size` The program tried to write an item other than a character constant across a record boundary.
F6700	`heap space limit exceeded` The program tried to open too many files at once. A file control block (FCB) must be allocated from the heap for each file opened, but no more heap space was available.
F6701	`scratch file name limit exceeded` The program exhausted the template used to generate unique scratch-file names.
F6980	`integer expected in format` An edit descriptor lacked a required integer value.
F6981	`initial left parenthesis expected in format` A format did not begin with a left parenthesis ().
F6982	`positive integer expected in format` A zero or negative integer value was used in a format. Negative integer values can appear only with the **P** edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.
F6983	`repeat count on nonrepeatable descriptor` One or more **BN**, **BZ**, **S**, **SS**, **SP**, **T**, **TL**, **TR**, **/**, ****, **$**, **:**, or apostrophe (') edit descriptors had repeat counts associated with them.
F6984	`integer expected preceding H, X, or P edit descriptor` An integer did not precede a (nonrepeatable) **H**, **X**, or **P** edit descriptor. The correct formats for these descriptors are *n***H**, *n***X**, and *k***P**, respectively, where *n* is a positive integer and *k* is an optionally signed integer.

Number	Run-Time Error Message
F6985	N or Z expected after B in format
	An illegal edit descriptor beginning with "B" was used.
	The only valid edit descriptors beginning with "B" are **BN** and **BZ**, used to specify the interpretation of blanks as nulls or zeros, respectively.
F6986	format nesting limit exceeded
	More than 16 sets of parentheses were nested inside the main level of parentheses in a format.
F6987	'.' expected in format
	A period did not appear between the *w* and *d* fields of a **D**, **E**, **F**, or **G** edit descriptor.
F6988	unexpected end of format
	An incomplete format was used.
	Improperly matched parentheses, an unfinished Hollerith (**H**) descriptor, or another incomplete descriptor specification can cause this error.
F6989	unexpected character in format
	A character that cannot be interpreted as part of a valid edit descriptor was used in a format.
F6990	M field exceeds W field in I edit descriptor
	The value of the *m* field specified in an **I** edit descriptor exceeded the value of the *w* field.
F6991	integer out of range in format
	An integer value specified in an edit descriptor was too large to represent as a 4-byte integer.
F6992	format not set by ASSIGN
	The format specifier in a **READ**, **WRITE**, or **PRINT** statement was an integer variable, but an **ASSIGN** statement did not properly assign it the statement label of a **FORMAT** statement in the same program unit.

E.4.2 Other Run-Time Error Messages

The following sections describe math run-time errors and general run-time errors. Math run-time errors are divided into low-level and function-level math errors.

Low-Level Math Errors

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions. These errors may also be detected by the floating-point emulator or alternate math library.

Using FORTRAN's default 8087/80287 control-word settings, the following exceptions are masked and do not occur:

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Inexact	Exception masked

See Appendix D, "Handling 8087/80287 Floating-Point Exceptions," for information on how to change the floating-point control word.

The following errors do not occur with code generated by the Microsoft FORTRAN Compiler or code provided in the standard Microsoft FORTRAN libraries:

```
square root
stack underflow
unemulated
```

The low-level math error messages, listed below, have the following format:

```
[[sourcefile(line) : ]] run-time error M61xx: MATH
- floating-point error: messagetext
```

The *sourcefile* and *line* where the error occurred appear only if the **/4Yb** option is used in compiling (or the **$DEBUG** metacommand is in effect).

Number	Low-Level Math Error Message
M6101	`invalid` An invalid operation occurred. This usually involves operating on NANs or infinities. This error terminates the program with exit code 129.
M6102	`denormal` A very small floating-point number was generated, which may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated upon. This error terminates the program with exit code 130.
M6103	`divide by 0` An attempt was made to divide by zero. This error terminates the program with exit code 131.
M6104	`overflow` An overflow occurred in a floating-point operation. This error terminates the program with exit code 132.
M6105	`underflow` An underflow occurred in a floating-point operation. (An underflow is normally masked so that the underflowing value is replaced with 0.0.) This error terminates the program with exit code 133.
M6106	`inexact` Loss of precision occurred in a floating-point operation. This exception is normally masked, since almost any floating-point operation can cause loss of precision. This error terminates the program with exit code 134.
M6107	`unemulated` An attempt was made to execute an 8087/80287 instruction that is invalid or is not supported by the emulator. This error terminates the program with exit code 135.

Number	Low-Level Math Error Message
M6108	square root

The operand in a square-root operation was negative. The FORTRAN intrinsic function **SQRT** does not generate this message; instead, **SQRT** gives a function-level DOMAIN error. This error terminates the program with exit code 136 (see error message M6201 below).

M6110	stack overflow

A floating-point expression caused a stack overflow on the 8087/80287 or emulator. Stack-overflow exceptions are trapped up to seven additional levels beyond the eight levels normally supported by the 8087/80287 processor. This error terminates the program with exit code 138.

M6111	stack underflow

A floating-point operation resulted in a stack underflow on the 8087/80287 or emulator. This error terminates the program with exit code 139.

Function-Level Math Errors

The function-level math error messages, listed below, appear when there are errors in the use of intrinsic functions. The error messages have the following format:

[[*sourcefile* (*line*) :]] run-time error M62*xx*: MATH
- *functionname*: *messagetext*

The *sourcefile* and *line* where the error occurred appear only if the **/4Yb** compiler option is used (or the **$DEBUG** metacommand is in effect).

Number	Function-Level Math Error Message
M6201	*functionname* : DOMAIN error

An argument to the given function was outside the domain of that function (that is, outside the legal set of input values for the function), as in the following examples:

```
SQRT(-1.0)
ACOS(-5.0)
```

Number	Function-Level Math Error Message
M6202	*functionname* : SING error This error refers to argument singularity. The given function was not properly defined for the value of its actual argument, although it may have been defined at nearby points, as in the following examples: LOG10(0.0) (0.0)**(-3.0)
M6203	*functionname* : OVERFLOW error The result of the given function or one of its intermediate computations was too large to be represented, as in the following example: EXP(25000.0)
M6204	*functionname* : UNDERFLOW error The result of the given function or one of its intermediate computations was too small to be represented. (This error is not currently supported; instead, the underflowing value goes to 0.0.)
M6205	*functionname* : TLOSS error A total loss of significance (precision) occurred, as in the following example: COS(1.0E30)
M6206	*functionname* : PLOSS error A partial loss of significance (precision) occurred. (This error is not currently supported; instead, the less-significant value is propagated to other computations or returned as the result.)

General Run-Time Error Messages

The following messages indicate general problems that may occur during program start-up, termination, or execution. These error messages have the following format:

‖*sourcefile* (*line*) : ‖ run-time error R6*xxx*
- *messagetext*

The *sourcefile* and the *line* where the error occurred appear only if the **/4Yb** compiler option is used to compile the program (or the **$DEBUG** metacommand is in effect). This additional information is not available for R6002, R6004, R6008, and R6009, which appear at start-up time.

Number	General Run-Time Error Message
R6000	stack overflow

Your program ran out of stack space. This can occur when a program uses a large amount of space for local data or temporary files or uses subprogram calls that are nested too deeply. The program is terminated with an exit code of 255.

To correct the problem, relink using the linker **/STACK** option to allocate a large stack, or relink using the compiler option **/F** *hexnum*. You can also compile your program with the **/Ge** compiler option to check for stack-overflow errors and modify the stack information in the executable-file header by using the **EXEMOD** program.

R6001 null pointer assignment

The contents of the **NULL** segment changed in the course of program execution. The **NULL** segment is a special low-memory location (starting at offset 0 in **DGROUP**) that is not normally used. If the contents of the **NULL** segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer (a memory address whose offset is 0 in the default data segment). Note that your program can contain null pointers without generating this message; the message appears only when you access a memory location through the null pointer.

This error does not cause your program to terminate; the error message is printed following the normal termination of the program. This error yields a nonzero exit code.

Number	General Run-Time Error Message

This message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

R6002 floating point not loaded

This error occurs when inadequate floating-point support has been loaded. The program terminates with exit code 255. Three situations can cause this error:

1. The program was compiled or linked with an option (such as **/FPi87**) that required an 8087 or 80287 coprocessor, but the program was run on a machine that did not have a coprocessor installed.

   To fix this problem, recompile the program with the **/FPi** option, relink with an emulator library (**LLIBFORE.LIB** or **MLIBFORE.LIB**), or install a coprocessor. (See Chapter 8, "Controlling Floating-Point Operations," for more information about these options and libraries.)

2. In a mixed-language program module that uses the C **scanf** or **printf** functions or their variants, a call to one of these functions included a floating-point format specification (such as **f**), but no floating-point values or variables appeared within the same C module. (The C compiler uses the presence of floating-point values and variables to determine whether or not to load floating-point conversion support.)

   To fix this problem, use a floating-point argument to correspond to the floating-point format specification in the **scanf** or **printf** call.

3. In a mixed-language program that uses both C and FORTRAN modules, a C library (**LLIBC.LIB** or **MLIBC.LIB**) was specified before a FORTRAN library (**LLIBFOR*x*.LIB or MLIBFOR*x*.LIB**) in the linking stage.

Number	General Run-Time Error Message
	To fix this problem, relink and specify the libraries in reverse order: the FORTRAN library followed by the C library. (See Section 11.3.12.3, "Linking Considerations," for more information.)
R6003	`integer divide by 0` An attempt was made to divide an integer by 0, giving an undefined result. Raising an integer 0 to a negative integer power may also cause this error. This error terminates the program with an exit code of 255.
R6004	`DOS 2.0 or later required` At start-up time, the DOS version was checked and found to be inadequate. Programs created by Version 4.0 of the Microsoft FORTRAN Compiler cannot run on versions of DOS prior to 2.0.
R6008	`not enough space for arguments` At start-up time, there was enough memory to load the program, but not enough room for the command-line arguments.
R6009	`not enough space for environment` At start-up time, there was enough memory to load the program, but not enough room for the environment table.

E.1 Linker Error Messages

This section lists and describes error messages generated by the Microsoft Overlay Linker, **LINK**. Limits imposed by the linker are described in Section E.12, "Compiler and Linker Limits."

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location `: error L1`*xxx*`:` *messagetext*

Nonfatal errors indicate problems in the executable file. **LINK** produces the executable file. Nonfatal error messages have the following format:

location : error L2*xxx*: *messagetext*

Warnings indicate possible problems in the executable file. **LINK** produces the executable file. Warnings have the following format:

location : warning L4*xxx*: *messagetext*

In these messages, *location* is the input file associated with the error, or LINK if there is no input file. If the input file is an **.OBJ** or **.LIB** file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

```
SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

The following error messages may appear when you link object files with the Microsoft Overlay Linker:

Number	Linker Error Message
L1001	*option* : option name ambiguous A unique option name did not appear after the option indicator (**/**). For example, the command `LINK /N main;` generates this error, since **LINK** cannot tell which of the three options beginning with the letter "N" was intended. (See Chapter 4 for more information about **LINK** options.)
L1002	*option* : unrecognized option name An unrecognized character followed the option indicator (**/**), as in the following example: `LINK /ABCDEF main;`
L1004	*option* : invalid numeric value An incorrect value appeared for one of the linker options. For example, a character string was given for an option that requires a numeric value.

Number	Linker Error Message
L1010	*option* : stack size exceeds 65536 bytes The size specified for the stack in the **/STACK** option of the **LINK** command was more than 65,536 bytes. (See Section 4.6.9 for more information about this option.)
L1007	*option* : interrupt number exceeds 255 A number greater than 255 was given as a value for the **/OVERLAYINTERRUPT** option.
L1008	*option* : segment limit set too high The limit on the number of segments allowed was set to greater than 1024 using the **/SEGMENTS** option. (See Section 4.6.11 for more information about this option.)
L1009	*option* : CPARMAXALLOC : illegal value The number specified in the **/CPARMAXALLOC** option was not in the range 1–65,535. (See Section 4.6.10 for more information about this option.)
L1020	no object modules specified No object-file names were specified to the linker.
L1021	cannot nest response files A response file occurred within a response file.
L1022	response line too long A line in a response file was longer than 127 characters.
L1023	terminated by user You entered CONTROL-C.
L1024	nested right parentheses The contents of an overlay were typed incorrectly on the command line. (See Section 4.7 for information about overlays.)

Number	Linker Error Message
L1025	nested left parentheses

The contents of an overlay were typed incorrectly on the command line. (See Section 4.7 for information about overlays.)

L1026 unmatched right parenthesis

A right parenthesis was missing from the contents specification of an overlay on the command line.

L1027 unmatched left parenthesis

A left parenthesis was missing from the contents specification of an overlay on the command line.

L1043 relocation table overflow

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references, where possible, and re-create the object module.

L1045 too many TYPDEF records

An object module contained more than 255 **TYPDEF** records. These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (**TYPDEF** is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 too many external symbols in one module

An object module specified more than the limit of 1023 external symbols.

Break the module into smaller parts. (See Section E.12.2 for more information on linker limits.)

Number	Linker Error Message
L1047	too many group, segment, and class names in one module
	The program contained too many group, segment, and class names.
	Reduce the number of groups, segments, or classes, and recreate the object file. (See Section E.12.2 for more information on linker limits.)
L1048	too many segments in one module
	An object module had more than 255 segments.
	Split the module or combine segments. (See Section E.12.2 for more information on linker limits.)
L1049	too many segments
	The program had more than the maximum number of segments. (The **/SEGMENTS** option specifies the maximum legal number; the default is 128.)
	Relink using the **/SEGMENTS** option with an appropriate number of segments. (See Section 4.6.11 for more information about this option, and Section E.12.2 for more information on linker limits.)
L1050	too many groups in one module
	LINK encountered more than 21 group definitions (**GRPDEF**) in a single module.
	Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS. See Section E.12.2 for more information on linker limits.)
L1051	too many groups
	The program defined more than 20 groups, not counting **DGROUP**.
	Reduce the number of groups. (See Section E.12.2 for more information on linker limits.)

Number	Linker Error Message
L1052	too many libraries

An attempt was made to link with more than 32 libraries.

Combine libraries, or use modules that require fewer libraries. (See Section E.12.2 for more information on linker limits.)

L1053 symbol table overflow

The program had more than 256K of symbolic information (such as public, external, segment, group, class, and file names).

Combine modules or segments and re-create the object files. Eliminate as many public symbols as possible. (See Section E.12.2 for more information on linker limits.)

L1054 requested segment limit too high

The linker did not have enough memory to allocate tables describing the number of segments requested. (The default is 128 or the value specified with the **/SEGMENTS** option.)

Try linking again using the **/SEGMENTS** option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells. (See Section E.12.2 for more information on linker limits.)

L1056 too many overlays

The program defined more than 63 overlays. (See Section E.12.2 for more information on linker limits.)

L1057 data record too large

A **LEDATA** record (in an object module) had more than 1024 bytes of data. This is a translator error. (**LEDATA** is a DOS term, which is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.

Number	Linker Error Message
L1070	segment size exceeds 64K
	A single segment contained more than 64K of code or data.
	Try compiling and linking using the large model. (See Chapter 9, "Working with Memory Models," for more information.)
L1071	segment _TEXT larger than 65520 bytes
	This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named **_TEXT** is linked using the **/DOSSEG** option of the **LINK** command. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.
L1072	common area longer than 65536 bytes
	The program had more than 64K of communal variables. This error cannot appear with object files generated by the Microsoft Macro Assembler, **MASM**. It occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.
L1080	cannot open list file
	The disk or the root directory was full.
	Delete or move files to make space.
L1081	out of space for run file
	The disk on which the **.EXE** file was being written was full.
	Free more space on the disk and restart the linker.
L1083	cannot open run file
	The disk or the root directory was full.
	Delete or move files to make space.
L1084	cannot create temporary file
	The disk or root directory was full.
	Free more space in the directory and restart the linker.

Number	Linker Error Message
L1085	`cannot open temporary file` The disk or the root directory was full. Delete or move files to make space.
L1086	`scratch file missing` An internal error has occurred. Note the circumstances of the problem and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1087	`unexpected end-of-file on scratch file` The disk with the temporary linker-output file was removed. (See Section 4.3, "Understanding LINK Memory Requirements," for more information about temporary linker-output files.)
L1088	`out of space for list file` The disk on which the listing file was being written was full. Free more space on the disk and restart the linker.
L1089	*filename* `: cannot open response file` **LINK** could not find the specified response file. This usually indicates a typing error.
L1090	`cannot reopen list file` The original disk was not replaced at the prompt. Restart the linker.
L1091	`unexpected end-of-file on library` The disk containing the library was probably removed. Replace the disk containing the library and run the linker again.

Number	Linker Error Message
L1093	`object not found` One of the object files specified in the linker input was not found. Restart the linker and specify the object file.
L1101	`invalid object module` One of the object modules was invalid. If the error persists after recompiling, please notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1102	`unexpected end-of-file` An invalid format for a library was encountered.
L1103	`attempt to access data outside segment bounds` A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1104	*filename* `: not valid library` The specified file was not a valid library file. This error causes **LINK** to abort.
L1113	`unresolved COMDEF; internal error` Note the circumstances of the failure and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1114	`file not suitable for /EXEPACK; relink without` For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image. Relink without the **/EXEPACK** option.

Number	Linker Error Message
L2001	`fixup(s) without data`

A **FIXUPP** record occurred without a data record immediately preceding it. This is probably a compiler error. (See the *Microsoft MS-DOS Programmer's Reference* for more information on **FIXUPP**.)

L2002 `fixup overflow near` *number* `in frame seg` *segname* `target seg` *segname* `target offset` *number*

The following conditions can cause this error:

- A group is larger than 64K.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link.
- An **EXTRN** declaration in an assembly-language source file appeared inside the body of a segment, as in the following example:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and re-create the object file. (For information about frame and target segments, refer to the *Microsoft MS-DOS Programmer's Reference*.)

Number	Linker Error Message
L2003	intersegment self-relative fixup
	An intersegment self-relative fixup is not allowed.
L2004	LOBYTE-type fixup overflow
	A **LOBYTE** fixup generated an address overflow. (See the *Microsoft MS-DOS Programmer's Reference* for more information.)
L2005	fixup type unsupported
	A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error. Note the circumstances of the failure and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L2011	'*name*' : NEAR/HUGE conflict
	Conflicting **NEAR** and **HUGE** attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.
L2012	'*name*' : array-element size mismatch
	A far communal array was declared with two or more different array-element sizes (for example, an array was declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the Microsoft Macro Assembler. It occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.
L2024	*name* : symbol already defined
	One of the special overlay symbols required for overlay support was defined by an object.
L2025	'*name*' : symbol defined more than once
	Remove the extra symbol definition from the object file.

Number	Linker Error Message
L2029	`unresolved externals`

One or more symbols were declared to be external in one or more modules, but they were not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:

```
EXIT in file(s):
 MAIN.OBJ (main.for)
OPEN in file(s):
 MAIN.OBJ (main.for)
```

The name that comes before `in file(s)` is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

L4012 `load-high disables EXEPACK`

The **/HIGH** and **/EXEPACK** options cannot be used at the same time.

L4015 `/CODEVIEW disables /DSALLOCATE`

The **/CODEVIEW** and **/DSALLOCATE** options cannot be used at the same time.

L4016 `/CODEVIEW disables /EXEPACK`

The **/CODEVIEW** and **/EXEPACK** options cannot be used at the same time.

L4020 *name* `: code-segment size exceeds 65500`

Code segments of 65,501 – 65,536 bytes in length may be unreliable on the Intel 80286 processor.

Number	Linker Error Message
L4021	no stack segment
	The program did not contain a stack segment defined with **STACK** combine type. This message should not appear for modules compiled with the Microsoft FORTRAN Compiler, but it could appear for an assembly-language module.
	Normally, every program should have a stack segment with the combine type specified as **STACK**. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the **STACK** combine type.
L4031	*name* : segment declared in more than one group
	A segment was declared to be a member of two different groups.
	Correct the source file and re-create the object files.
L4050	too many public symbols
	The **/MAP** option was used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (more than 3072 symbols by default).
	Relink using **/MAP:***number*. The linker produces an unsorted listing of the public symbols.
L4051	*filename* : cannot find library
	The linker could not find the specified file.
	Enter a new file name, a new path specification, or both.
L4053	VM.TMP : illegal file name; ignored
	VM.TMP appeared as an object-file name.
	Rename the file and rerun the linker.
L4054	*filename* : cannot find file
	The linker could not find the specified file.
	Enter a new file name, a new path specification, or both.

E.6 LIB Error Messages

Error messages generated by the Microsoft Library Manager, **LIB**, have one of the following formats:

{*filename* | LIB} : fatal error U1*xxx*: *messagetext*
{*filename* | LIB} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, **LIB** prints a warning and continues operation. In some cases errors are fatal and **LIB** terminates processing. **LIB** may display the following error messages:

Number	LIB Error Message
U1150	page size too small The page size of an input library was too small, which indicates an invalid input **.LIB** file.
U1151	syntax error : illegal file specification A command operator such as a minus sign (−) was given without a following module name.
U1152	syntax error : option name missing A forward slash (**/**) was given without an option following it.
U1153	syntax error : option value missing The **/PAGESIZE** option was given without a value following it.
U1154	option unknown An unknown option was given. Currently, **LIB** only recognizes the **/PAGESIZE** option.
U1155	syntax error : illegal input The given command did not follow correct **LIB** syntax as specified in Chapter 5, "Managing Libraries."

Number	LIB Error Message
U1156	`syntax error` The given command did not follow correct **LIB** syntax as specified in Chapter 5, "Managing Libraries."

U1157	`comma or new line missing`
	A comma or carriage return was expected in the command line but did not appear. This may indicate an inappropriately placed comma, as in the following line:
	`LIB math.lib,-mod1+mod2;`
	The line should have been entered as follows:
	`LIB math.lib -mod1+mod2;`

U1158	`terminator missing` Either the response to the "Output library" prompt or the last line of the response file used to start **LIB** did not end with a carriage return.
U1161	`cannot rename old library` **LIB** could not rename the old library to have a **.BAK** extension because the **.BAK** version already existed with read-only protection. Change the protection on the old **.BAK** version.
U1162	`cannot reopen library` The old library could not be reopened after it was renamed to have a **.BAK** extension.
U1163	`error writing to cross-reference file` The disk or root directory was full. Delete or move files to make space.
U1170	`too many symbols` More than 4609 symbols appeared in the library file.

Number	LIB Error Message
U1171	`insufficient memory` **LIB** did not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.
U1172	`no more virtual memory` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1173	`internal failure` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1174	`mark: not allocated` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1175	`free: not allocated` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1180	`write to extract file failed` The disk or root directory was full. Delete or move files to make space.
U1181	`write to library file failed` The disk or root directory was full. Delete or move files to make space.
U1182	*filename* `: cannot create extract file` The disk or root directory was full, or the specified extract file already existed with read-only protection. Make space on the disk or change the protection of the extract file.

Number	LIB Error Message
U1183	`cannot open response file` The response file was not found.
U1184	`unexpected end-of-file on command input` An end-of-file character was received prematurely in response to a prompt.
U1185	`cannot create new library` The disk or root directory was full, or the library file already existed with read-only protection. Make space on the disk or change the protection of the library file.
U1186	`error writing to new library` The disk or root directory was full. Delete or move files to make space.
U1187	`cannot open VM.TMP` The disk or root directory was full. Delete or move files to make space.
U1188	`cannot write to VM` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1189	`cannot read from VM` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1200	*name* `: invalid library header` The input library file had an invalid format. Either it was not a library file, or it had been corrupted.

Number	LIB Error Message
U1203	*name* : invalid object module near *location* The module specified by *name* was not a valid object module.
U4150	*modulename* : module redefinition ignored A module was specified to be added to a library but a module with the same name was already in the library. Or, a module with the same name was found more than once in the library.
U4151	*symbol* (*modulename*) : symbol redefinition ignored The specified symbol was defined in more than one module.
U4152	*filename* : cannot create listing The directory or disk was full, or the cross-reference-listing file already existed with read-only protection. Make space on the disk or change the protection of the cross-reference-listing file.
U4153	*number* : page size too small; ignored The value specified in the **/PAGESIZE** option was less than 16.
U4155	*modulename* : module not in library; ignored The specified module was not found in the input library.
U4156	*libraryname* : output-library specification ignored An output library was specified in addition to a new library name. For example, specifying `LIB new.lib+one.obj,new.lst,new.lib` where `new.lib` does not already exist causes this error.
U4157	*filename* : cannot access file **LIB** was unable to open the specified file.

Number	LIB Error Message
U4158	*libraryname* : invalid library header ; file ignored The input library had an incorrect format.
U4159	*filename* : invalid format *hexnumber* ; file ignored The signature byte or word *hexnumber* of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX archive.

E.7 MAKE Error Messages

Error messages displayed by the Microsoft Program Maintenance Utility, **MAKE**, have one of the following formats:

{*filename* | MAKE} : fatal error U1*xxx*: *messagetext*
{*filename* | MAKE} : warning U4*xxx*: *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. If possible, **MAKE** prints a warning and continues operation. In some cases, errors are fatal and **MAKE** terminates processing. **MAKE** generates the following error messages:

Number	MAKE Error Message
U1001	macro definition larger than *number* A single macro was defined to have a value string longer than the number stated, which is the maximum. Try rewriting the **MAKE** description file to split the macro into two or more smaller ones.

U1002	infinitely recursive macro
	A circular chain of macros was defined, as in the following example:
	`A=$(B)`
	`B=$(C)`
	`C=$(A)`

Number	MAKE Error Message
U1003	`out of memory` **MAKE** ran out of memory for processing the **MAKE** description file. Try to reduce the size of the **MAKE** description file by reorganizing or splitting it.
U1004	`syntax error : macro name missing` The **MAKE** description file contained a macro definition with no left side (that is, a line beginning with =).
U1005	`syntax error : colon missing` A line that should be an outfile/infile line lacked a colon indicating the separation between outfile and infile. **MAKE** expects any line following a blank line to be an outfile/infile line.
U1006	*targetname* `: macro expansion larger than` *number* A single macro expansion, plus the length of any string to which it may be concatenated, was longer than the number stated. Try rewriting the **MAKE** description file to split the macro into two or more smaller ones.
U1007	`multiple sources` An inference rule was defined more than once.
U1008	*name* `: cannot find file or directory` The file or directory specified by *name* could not be found.
U1009	*command* `: argument list too long` A command line in the **MAKE** description file was longer than 128 bytes, which is the maximum that DOS allows. Rewrite the commands to use shorter argument lists.

Number	MAKE Error Message
U1010	*filename* : permission denied The file specified by *filename* was a read-only file.
U1011	*filename* : not enough memory Not enough memory was available for **MAKE** to execute a program.
U1012	*filename* : unknown error Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1013	*command* : error *errcode* One of the programs or commands called in the **MAKE** description file returned with a nonzero error code.
U4000	*filename* : target does not exist This usually does not indicate an error. It warns the user that the target file does not exist. **MAKE** executes any commands given in the block description, since in many cases the outfile will be created by a later command in the **MAKE** description file.
U4001	dependent *filename* does not exist; target *filename* not built **MAKE** could not continue because a required infile did not exist. Make sure that all named files are present and that they are spelled correctly in the **MAKE** description file.
U4013	*command* : error *errcode* (ignored) One of the programs or commands called in the **MAKE** description file returned with a nonzero error code, and **MAKE** was run with the **/I** option. **MAKE** ignores the error and continues.

Number	MAKE Error Message
U4014	`usage : make [/n] [/d] [/i] [/s] [name=value ...] file` **MAKE** has not been invoked correctly. Try entering the command line again with the syntax shown in the message.

E.8 EXEPACK Error Messages

Error messages generated by the Microsoft EXE File Compression Utility, **EXEPACK**, have one of the following formats:

{*filename* | EXEPACK} : fatal error U1*xxx*: *messagetext*
{*filename* | EXEPACK} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility.

If possible, **EXEPACK** prints a warning and continues operation. In some cases, errors are fatal and **EXEPACK** terminates processing. Fatal errors have an exit code of 1. **EXEPACK** generates the following error messages:

Number	EXEPACK Error Message
U1100	`out of space on output file` The disk or root directory is full. Delete or move files to make space.
U1101	*filename* : `file not found` The file specified by *filename* could not be found.
U1102	*filename* : `permission denied` The file specified by *filename* was a read-only file.
U1103	`cannot pack file onto itself` It is illegal to specify the same file for both input and output.

Number	EXEPACK Error Message
U1104	`usage : exepack <infile> <outfile>` The **EXEPACK** command line was not specified properly. Try again using the syntax shown.
U1105	`invalid .EXE file; bad header` The given file was not an executable file, or it had an invalid file header.
U1106	`cannot change load-high program` When the minimum allocation value and the maximum allocation value are both 0, the file cannot be compressed.
U1107	`cannot pack already-packed file` The file specified for **EXEPACK** had already been packed using **EXEPACK**.
U1108	`invalid .EXE file; actual length less than reported` The second and third fields in the file header indicated a file size greater than the actual size.
U1109	`out of memory` The **EXEPACK** utility did not have enough memory to operate.
U1110	`error reading relocation table` The file could not be compressed because the relocation table could not be found or was invalid.
U1111	`file not suitable for packing` The packed load image of the specified file was larger than the unpacked load image, so the file could not be packed.
U1112	*filename* `: unknown error` An unknown system error occurred while the specified file was being read or written. Try running **EXEPACK** again.

Number	EXEPACK Error Message
U4100	`omitting debug data from output file` **EXEPACK** strips symbolic debug information from the input file before packing.

You may also encounter DOS error messages if the **EXEPACK** program cannot read from, write to, or create a file.

E.9 EXEMOD Error Messages

Error messages from the Microsoft EXE File Header Utility, **EXEMOD**, have one of the following formats:

{*filename* | EXEMOD} : fatal error U1*xxx*: *messagetext*
{*filename* | EXEMOD} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, **EXEMOD** prints a warning and continues operation. In some cases, errors are fatal and **EXEMOD** terminates processing. **EXEMOD** generates the following error messages:

Number	EXEMOD Error Message
U1050	`usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]` The **EXEMOD** command line was not specified properly. Try again using the syntax shown. Note that the option indicator can be either a slash (**/**) or a dash (**-**). The single brackets ([ ]) in the error message indicate that your choice of the item within them is optional.
U1051	`invalid .EXE file : bad header` The specified input file is not an executable file or has an invalid file header.
U1052	`invalid .EXE file : actual length less than reported` The second and third fields in the input-file header indicate a file size greater than the actual size.

Number	EXEMOD Error Message
U1053	`cannot change load-high program` When the minimum allocation value and the maximum allocation value are both 0, the file cannot be modified.
U1054	`file not .EXE` **EXEMOD** automatically appends the **.EXE** extension to any file name without an extension; in this case, no file with the given name and an **.EXE** extension could be found.
U1055	*filename* `: cannot find file` The file specified by *filename* could not be found.
U1056	*filename* `: permission denied` The file specified by *filename* was a read-only file.
U4050	`packed file` The given file was a packed file. This is a warning only.
U4051	`minimum allocation less than stack; correcting minimum` If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. This is a warning message only; the modification is still performed.
U4052	`minimum allocation greater than maximum; correcting maximum` If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the modification is still performed. **EXEMOD** will still modify the file. The values shown if you ask for a display of DOS header values will be the values after the packed file is expanded.

E.10 SETENV Error Messages

Messages generated by the Microsoft Environment Expansion Utility, **SETENV**, have the following format:

{*filename* | SETENV} : fatal error U1*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. **SETENV** generates the following error messages:

Number	SETENV Error Message
U1080	usage : setenv <command.com> [envsize] The command line was not specified properly. This usually indicates that the wrong number of arguments was given. Try again with the syntax shown in the message.
U1081	unrecognizable COMMAND.COM The **COMMAND.COM** file was not one of the accepted versions (IBM PC-DOS, Versions 2.0, 2.1, 2.11, 3.0, and 3.1).
U1082	maximum for Version 3.1 : 992 The user specified a file that was recognized as **COMMAND.COM** for IBM PC-DOS, Version 3.1, and gave an environment size greater than 992 bytes, the maximum allowed for that version.
U1083	maximum environment size : 65520 The environment size specified was greater than 65,520 bytes, the maximum size allowed.
U1084	minimum environment size : 160 The environment size specified was less than 160 bytes, the minimum size allowed.
U1085	*filename* : cannot find file The specified file was not found, or it was a directory or some other special file.

Number	SETENV Error Message
U1086	*filename* : permission denied The specified file was a read-only file.
U1087	*filename* : unknown error An unknown system error occurred while the specified file was being read or written. Try running **SETENV** again.

E.11 ERROUT Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

command line error U1*xxx*: *messagetext*
execution error U2*xxx*: *messagetext*

ERROUT generates the following error messages:

Number	ERROUT Error Message
U1251	no arguments No arguments were specified to **ERROUT**.
U1252	bad command line switch An option other than **/f** was given on the **ERROUT** command line.
U1253	missing file name The **/f** option was given on the **ERROUT** command line without a file name.
U1254	missing command No *command* was given on the **ERROUT** command line.

Number	ERROUT Error Message
U2251	`cannot open file` **ERROUT** could not open the given *stderrfile*.
U2252	`cannot redirect standard error` The *stderrfile* given on the **ERROUT** command line could not be used for standard error output.
U2253	`command failed` The *command* given on the **ERROUT** command line failed.

E.12 Compiler and Linker Limits

This section discusses the limits imposed by the Microsoft FORTRAN Compiler and the Microsoft Overlay Linker.

E.12.1 Compiler Limits

This section summarizes limits imposed by the Microsoft FORTRAN Compiler (for example, the maximum length of an identifier) and suggests programming strategies for avoiding these limits.

To operate the Microsoft FORTRAN Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table E.1 summarizes the limits imposed by the Microsoft FORTRAN Compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.1
Limits Imposed by the Microsoft FORTRAN Compiler

Program Item	Maximum Limit
Actual arguments	Number per subprogram: approximately 64
Character constants	Length: approximately 1900 bytes
Names	Length: 6 bytes (default) or 31 bytes (if **/4Yt** is used in compiling or if **$NOTRUNCATE** is in effect); additional characters are discarded
	Internal, length: 40 bytes; number per module: 20,000 names
Simple variables	Number of simple variables per subprogram: approximately 3500 (depending on lengths of the variable names)
Statements	Levels of nesting: approximately 40 levels
ENTRY statements	Number per subroutine: 32,000 statements
FORMAT statements	Number per module: 20,000
	Format length: approximately 1900 characters
	Memory limitations: in medium-model programs, no more than 64K internal formats in the default data segment
	Number of errors per statement: 10 errors
GOTO statements (assigned)	Number per subroutine: 64
INTEGER items	Size: 128 bytes for a string of digits
Include files	Levels of nesting: 10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

During compilation, large programs are most often limited in the number of identifiers allowed in any one source file. They are also occasionally limited by the complexity of the program or one of its statements.

E.12.1.1 Limits on Number of Names

The Microsoft FORTRAN Compiler limits the number of names you can use in a source program. The compiler creates symbol-table entries for the names declared in source programs. Symbol-table entries are created for the following objects:

- The program
- Subroutines and functions declared or referenced in the program unit
- Common blocks and variables
- Statement functions
- Formal parameters
- Local variables

Common variables, statement functions, formal parameters, and local variables are required only while the subroutine or function that contains them is being compiled. These names are discarded at the end of the subroutine, and the space they used is made available for other names. Hence, you can create much bigger programs by splitting up your code into more subroutines and functions so that the space for "local" names can be shared. You can also place the subroutines and functions into their own files and compile them separately, since this usually reduces the number of names in groups being used per module.

E.12.1.2 Limits on Complicated Expressions

The compiler may run out of memory when it encounters any of the following:

- A deeply nested statement or expression
- A large number of error messages
- A large block of specification statements (**EQUIVALENCE** statements in particular)

Usually, if Pass 1 runs successfully on a program without running out of memory, Pass 2 will also run successfully, except for complicated basic blocks. A basic block is defined as follows:

- A sequence of statements with no labels or other breaks
- A sequence of statements containing long expressions or parameter lists (especially including I/O statements or character expressions)

Pass 2 makes a smaller number of symbol-table entries than Pass 1 (for example, for the program, subroutines, and functions declared or referenced in the program unit, for common blocks, and for many of the transcendental functions called in a program). If Pass 2 runs out of memory, it displays a line-number reference and one of the following messages:

```
out of heap space

expression too complex, please simplify
```

If a particularly long expression or parameter list appears near this line, break up the expression or parameter list by assigning parts of the expression to local variables or by using multiple **WRITE** statements. If this does not work, add labels to statements to break the basic block.

E.12.1.3 Limits on Character Expressions

Use the following programming strategies to avoid compiler limitations when initializing or assigning values to large character variables or array elements:

- Use smaller pieces
- Use substrings
- Use **EQUIVALENCE** statements to assign values to a character array

To avoid compiler limitations on character expressions, assign pieces of the character value to smaller variables or substrings. Just having nonconstants in the expression causes more of the expression to be evaluated at run time instead of at compile time, thus avoiding the 1900-character compile-time limit on constants.

E.12.2 Linker Limits

Table E.2 summarizes the limits imposed by the linker. If you encounter one of these limits, you must adjust your program so that the linker can accommodate it.

Table E.2

Limits Imposed by the Microsoft Overlay Linker

Item	Limit
Symbol table	256K
Load-time relocations	Default is 32K. If **/EXEPACK** is used, the maximum is 512K.
Public symbols	The range 7700 – 8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defines **DGROUP** so the effective maximum is 20.
Overlays	63
Segments	128 by default; however, this maximum can be set as high as 1024 by using the **/SEGMENTS** option of the **LINK** command.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64K

E.12.2 Linker Limits

Table E.2 summarizes the limits imposed by the linker. If you encounter one of these limits, you must adjust your program so that the linker can accommodate it.

Table E.2

Limits Imposed by the Microsoft Overlay Linker

Item	Limit
Symbol table	256K
Load-time relocations	Default is 32K. If **/EXEPACK** is used, the maximum is 512K.
Public symbols	The range 7700 – 8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defines **DGROUP** so the effective maximum is 20.
Overlays	63
Segments	128 by default; however, this maximum can be set as high as 1024 by using the **/SEGMENTS** option of the **LINK** command.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64K

Index

Index

& (ampersand), LIB command symbol, 155
* (asterisk), LIB command symbol, 152, 157, 160
| (bar), 11
{ } (braces), 11
[ ] (brackets), 11
: (colon), LINK command, 114
, (comma)
  LIB command symbol, 150
  LINK command symbol, 111
- (dash)
  EXEMOD option character, 185
  FL option character, 58
$ (dollar sign), Versions 4.0 and 3.3, differences, 321
... (dots), 12
/ (forward slash)
  EXEMOD option character, 185
  FL option character, 58
  LINK option character, 121
– (minus sign), LIB command symbol, 151, 154, 157, 159
– * (minus sign-asterisk), LIB command symbol, 152, 160
– + (minus sign-plus sign), LIB command symbol, 152, 153, 159
+ (plus sign)
  LIB command symbol
    appending object files, 157, 158, 159
    combining libraries, 160
    Intel, XENIX files, used with, 147
    specifying library, 154
  LINK command symbol, 113, 116
; (semicolon)
  LIB command symbol, 149, 150, 156, 161
  LINK command symbol, 111, 115, 116
" " (quotation marks), 12
' (single right quotation mark), 10
_ (underscore)
  C names, used in, 272
_ (underscore) *(continued)*
  FORTRAN 4.0 names, used in, 321
__ (underscores), 265

/4I2 and /4I4 options (FL), 90, 237
/4Nb option (FL), 86
/4Y6 and /4N6 options (FL), 91
/4Yb and /4Nb options (FL), 217
/4Yb option (FL), 44, 86, 237
/4Yd and /4Nd options (FL), 88
/4Yd option (FL), 44
/4Yf and /4Nf options (FL), 44, 91
/4Ys and /4Ns options (FL), 91
/4Yt and /4Nt options (FL), 91
/4Yt option (FL), 272
80186/80188 processor, 40, 96
80286 processor, 40, 96
8087/80287 coprocessor
  math package, 193
  suppressing use of, 204

Addresses
  far
    code, 222
    defined, 211, 212
    large and huge models, 220
    subprogram calls, 213
  huge
    defined, 211, 212, 214
    huge arrays, 217
    huge model, 225
  near
    default data segment, 214, 216
    defined, 211, 212
    medium model, 226
    subprograms, 223
  passing, 268
  segment start, 139
Affine mode, 361
/AH option (FL), 59, 225
/AL option (FL), 59, 224
ALIAS attribute, 272

Alignment types, 138, 140, 259
Alternate math library, 199
/AM option (FL), 59, 226
Ampersand (&), LIB command symbol, 155
ANSI X3.9-1978 full-language standard
  Version 4.0 changes for, 307
Apostrophe ('), described, 10
Archives, XENIX, 147, 160
Argument-passing conventions
  C, 262
  FORTRAN, 260
Arguments
  See also Passing arguments
  C, handling in FORTRAN, 279
  calling conventions, default, 269
  conversion, 260
  first, address on stack, 263
  FL options, 58
  LINK options, 122
  listing options (FL), 68
  passing
    by reference, 267, 268, 279
    by value, 220, 267, 268
    medium model, 226
    varying numbers of, 271
  procedural, 293
  pushing, 260
  removing from stack, 261, 265
  varying numbers of, 267
Arrays
  addressing, 218
  adjustable size, 218
  assumed size, 218
  C, passing in, 288
  constant use, minimizing in declarations, 242
  declarations, Versions 4.0 and 3.3, 309
  declaring for efficient compilation, 242, 243
  EQUIVALENCE statements, used in, 242
  formal arguments, used as, 219
  FORTRAN and C, 290
  FORTRAN procedures, used in, 288
  huge
    addressing, 217
    mixed languages, 288
Arrays *(continued)*
  mixed-language programming, 288
  subscripts, Versions 4.0 and 3.3, differences, 323
Assembly-language routines
  assembling, 254
  entering, 262
  exiting, 264
  optimizing, 252
  program example, 247
Assembly-listing files
  creating, 67
  extensions, 68
  format, 77
Asterisk (*), LIB command symbol, 152, 160
Attributes
  ALIAS, 272
  array declarations, Versions 4.0 and 3.3, differences, 309
  C
    argument-passing conventions, 262, 270
    naming conventions, 272
    removing arguments from stack, 265
    varying numbers of arguments, 271
  calling conventions, specifying, 268
  EXTERN, 294
  FAR
    adjustable-size arrays, 219
    arguments in medium model, 220
    assumed-size arrays, 219
    default data segment, 214, 216, 217
    effects, 230
    huge model, 225
    library routines, used with, 232
    medium model, 226, 227
    using, 228
  HUGE
    alternative to huge model, 225
    common blocks, 219
    default data segment, 214, 216, 217
    effects, 230
    large model, 225
    library routines, used with, 232
    medium model, 220, 226, 227

Attributes *(continued)*
HUGE *(continued)*
using, 228
lack of portability, 228
mixed-language programming, 269
NEAR
adjustable-size arrays, 219
alternative to medium model, 227
assumed-size arrays, 219
common blocks, 220
declaring subprograms with, 222
effects, 229
huge model, 225
large and huge models, 216, 220, 226
library routines, used with, 232
subprograms, 223
VALUE, 220
VARYING, 271
AUTOEXEC.BAT file, 21, 34, 38
AUX, 70

Back-up procedures, 17
Bar (|), 11
Batch files
exit codes, 338
FL command, converting for, 318
SET and PATH, 38
using, 47
BEGDATA class name, 131
Bibliography, 13
Blanks in formatted files, Versions 4.0 and 3.3, differences, 309
BLOCKSIZE option, 217, 241
Bold type, 10
Boolean types, 282
BP register, 262, 264
Braces ({ }), 11
Brackets ([ ]), 11
BSS class name, 131
_BSS segment, 255, 256, 258

C
See also Mixed-language programming
attribute
argument-passing conventions, 262, 270
naming conventions, 272

C *(continued)*
attribute *(continued)*
removing arguments from stack, 265
varying numbers of arguments, 271
calling conventions, 263
/c option (FL), 43, 63
Calling conventions
C, 263
FORTRAN, 263
FORTRAN and C, 267
mixed-language programming, 267, 268
Canonical frame number. *See* Frame number
Capital letter
See also Case significance
notation, 10
small, 13
Case significance
C names, 272
LINK, 118, 123, 127
c_common segment, 256, 258
Character constants, maximum size, 323, 482
Character expressions, limits on, 484
Character types
mixed-language programming, 282
variables as format specifiers, 240
Class names
BEGDATA, 131
BSS, 131
CODE, 131
linking procedure, used in, 139
STACK, 131, 259
Class types, 139
Classes
$COMMQQ, 257
defined, 213
FAR_BSS, 257
FAR_DATA, 257
HUGE_BSS, 257
table, 259
/CO option. *See* LINK options, /CODEVIEW
CODE class name, 131
Code size
limits, 222
optimizing, 97
CodeView exit codes, 339

/CODEVIEW option (LINK), 134
Colon (:), LINK command, 114
Combine
  classes, 259
  types
    COMMON, 140
    LINK, 140
    PRIVATE, 140
    PUBLIC, 140
    STACK, 140
Comma (,)
  LIB command symbol, 150
  LINK command symbol, 111
Command line
  error messages, 368
  FL, 52, 109
  LIB, 149
  LINK, 111
Commands, DOS
  IF ERRORLEVEL, 48, 86
  PATH, 17, 38
  SET, 17, 37, 38
Common blocks
  formal arguments, used as, 219
  large and huge models, used in, 219
  medium model, used in, 220, 226
  memory allocation, 219
  restrictions, 219
$COMMQQ class, 257, 294
Compatibility
  8087/80287 library, 205
  emulator library, 205
  floating-point options, 203
  Versions 3.2 and 3.3, using SETUP for, 33
Compilation, conditional, 93, 329
Compilation error messages, 372
Compiler
  documentation, 5
  error messages
    See also Error messages
    categories, 372
    compilation, 372
    correctable, 373, 426
    fatal, 372, 373
    identifying, 85
    redirecting, 85
    warning, 373, 426
  exit codes, 86
  files, default directory, 20
Compiler *(continued)*
  limits, 481
  mixed-language programming, versions required for, 266
  options. *See* FL options
  system requirements, 4
Complex numbers, 291
CON, 70
CONFIG.SYS file
  buffers parameter, 40
  files parameter, 39
  SETUP, 35, 39
Consistency checking (LIB), 150, 161
CONST segment, 256, 258
Controlling
  data loading, 132
  executable-file loading, 133
  LINK, 121
  segments, 130
  stack size, 128
Coprocessor
  8087/80287, 193
  suppressing use of, 204
Correctable error messages, 373, 426
/CP option. *See* LINK options, /CPARMAXALLOC
/CPARMAXALLOC option (LINK), 129
Cross-reference listing (LIB), 152, 161
CRT0.OBJ. *See* Start-up routine
CS register, 212, 213, 262, 266

/D option (MAKE), 173
_DATA segment, 234, 256, 257, 258
Data segments
  data threshold, 232
  default
    contents, 216
    _DATA, 256
    defined, 213
    limits, 216
    naming, 234
    near addresses, 214
    threshold, setting, 232
  loading, 132
  naming, 234
Data threshold, setting, 232
Data types
  equivalent, FORTRAN and C, 278
  mixed-language programming, 278

Data types *(continued)*
Version 4.0, new, 330
$DEBUG metacommand, 86, 217, 238, 357
Debugging, preparing for
described, 94
LINK (/CODEVIEW option), 134
$DECLARE metacommand, 88, 334
Declaring procedures, mixed-language programming, 276
Default
data segment
address of, 216
contents, 216, 256
data threshold, 232
defined, 213, 214
limits, 216, 226
object file, 220
libraries
ignoring, 121, 127
object file, 119
responses
LIB, 156
LINK, 115
DEMOEXEC.FOR, 296
DEMO.FOR, 21, 25
Denormal
exception, 448
numbers, 194
propagating, 363
Description file, 166
Device names, 70
DGROUP
allocating memory below, 132
NULL segment, 256
segment order, 131
segments, 258
DI register, 262, 264, 265
Differences, Versions 4.0 and 3.3
4.0 and 3.3 modules, mixing, 316
ANSI full-language standard, 307
binary direct files, 311
blanks in formatted files, 309
compatibility
library, 315
object, 315
source, 309
compiling and linking, 317, 318
DO-loop ranges, 315
exponentiation, 312
Differences, Versions 4.0 and 3.3 *(continued)*
floating-point precision, 311
language changes, 317, 320, 330
list-directed output, 314
MODE and STATUS options, 310
run-time libraries, 320
scratch-file names, 310
SETUP, linking libraries with, 310
Direction flag, 265
Disabling optimization, 94, 97
Disks
backing up, 17
DOS, 21, 25, 39
Disks, boot. *See* Disks, DOS
Disks, compiler package
contents, 18
Learning Microsoft CodeView, 20, 24
Setup
3-1/2-inch disks, used with, 25
5-1/4-inch disks, used with, 21
PACKING.LST, 18
README.DOC, 24
SETUP, used with, 20
Utilities, Source Code, and Microsoft CodeView, 20, 21, 25
Disks, system. *See* Disks, DOS
/DO option. *See* LINK options, /DOSSEG
$DO66 metacommand, 87
Documentation, compiler, 5
Dollar sign ($), Versions 4.0 and 3.3, differences, 321
DO-loop ranges, Versions 4.0 and 3.3, differences, 315
DOS
commands
IF ERRORLEVEL, 48, 86
PATH, 17, 34
SET, 17, 34, 37
program header, 186
/DOSSEG option (LINK), 131, 254
Dots (...), 12
/DS option. *See* LINK options, /DSALLOCATE
DS register
assembly-language routine, 262, 266
default data segment, 213, 216
described, 132
DGROUP, 258

DS register *(continued)*
near addresses, 213
near, far, and huge addresses, 212
/DSALLOCATE option (LINK), 132

/E option. *See* LINK options, /EXEPACK
Edit descriptors
Tc, 240
TLc, 240
Z, 333
Edit lists, 240
Ellipsis dots (...), 12
EMOEM.ASM, 206
Emulator
described, 194
function calls, 199
in-line instructions, 199
library, 199
Entry sequence, assembly-language, 262
ENTRY statement, maximum number per subroutine, 482
Environment
batch files, setting up with, 47
table
enlarging, 188
limits, 38
Environment variables
assigning, 37
defined, 34
INCLUDE, 35, 36, 81
LIB, 36, 40, 119
NO87, 204
PATH
defined, 35
described, 36
DOS commands, 17
RAM disk, used with, 40
SET, used with, 38
search paths, 37
SET, 17, 37
SETUP, 34
TMP, 35, 37
EQUIVALENCE statement, 242, 290
Error messages
command-line, 368
compiler
compilation, 372
Error messages *(continued)*
compiler *(continued)*
correctable, 373, 426
defined, 372
fatal, 372, 373
redirecting, 85
warning, 373, 426
ERROUT, 480
EXEMOD, 477
EXEPACK, 475
format
compiler, 373
run-time, 434
LIB, 467
LINK, 454
MAKE, 472
mixed-language programming, 300
removing text during SETUP, 32, 237
run-time
floating-point exceptions, 448
redirecting, 190
run-time library, 435
SETENV, 479
warning messages, setting level of, 89
Errorlevel codes. *See* Exit codes
Errors, maximum number per statement, 482
ERROUT
described, 190
error messages, 480
exit codes, 340
ES register, 212, 258, 266
Exception handling
control word, 357
dividing by zero, 358
invalid operation, 358
overflow and underflow, 358
precision, 358
status byte, 357
Executable
files
changing headers, 185
compressing, 183
extensions, 65, 66, 112
FL command, used with, 53
loading, 133
naming, default, 65, 104, 112
naming with FL, 65
naming with LINK, 112

Executables *(continued)*
files *(continued)*
packing, 125
specifying with LINK, prompts, 115
specifying with LINK, response file, 116
image, 138
Execution time, optimizing, 97
EXEMOD
default stack size, changing, 128
described, 185
error messages, 477
exit codes, 340
/H option, 186
/MAX option, 186
maximum allocation, changing, 129
/MIN option, 186
option character
dash (-), 185
forward slash (/), 185
/STACK option, 186
stack size, setting, 216, 221
EXEPACK
command line, 184
described, 183
error messages, 475
exit codes, 340
symbolic debug information, stripping, 184
/EXEPACK option (LINK), 125
Exit codes
CodeView, 339
DOS, 338
error level
0, 1 codes, 337
2, 4 codes, 339
FL, 48, 339
FORTRAN programs, 341
using, 337
Exit sequence, assembly language, 264
Exponentiation exceptions, 312
Expressions, compiler limits, avoiding, 483
Extensions
default, LINK, 118
executable files, 65, 66, 112
libraries, 118, 147, 149, 150
map files, 68, 113, 118, 126
object files, 64, 112, 118
object-listing files, 68
Extensions *(continued)*
source-listing files, 68
source/object-listing files, 68
EXTERN attribute, 294

/F option (FL), 102, 128
/Fa option (FL), 67, 247, 262
Far addresses
code, 222
data threshold, 232
defined, 211, 212
large and huge models, 220
subprogram calls, 213
FAR attribute
adjustable-size arrays, 219
assumed-size arrays, 219
default data segment, 214, 216, 217
effects, 230
huge model, 225
library routines, used with, 232
medium model, arguments in, 220, 226, 227
using, 228
FAR_BSS class, 257
FAR_DATA class, 257
Fatal error messages, 372, 373
/Fc option (FL), 67, 262
/Fe option (FL), 65
File names
See also Naming
scratch files, Versions 4.0 and 3.3, differences, 310
specifying on the command line, 56
File-control blocks (FCBs), 217
File-name conventions, LINK, 118
Files
assembly listing, 67, 76
AUTOEXEC.BAT, 21, 34, 38
batch, 38, 47
compiler, 20
CONFIG.SYS, 35, 39, 40
DEMO.FOR, 45
executable
environment variables, 37
naming, default, 104
naming with FL, 65
naming with LINK, 112
FL.EXE, 34

Files *(continued)*
FORTRAN
access modes, 345
binary direct, 311, 352
binary sequential, 351
data formats, 345
formatted direct, 347
formatted sequential, 345
record structure, 345
unformatted direct, 350
unformatted sequential, 348
include, 37
library, 37
locating, 34
map
creating, 69, 126
default names, 68
frame numbers, 139
listing formats, 79
/MAP option (LINK), 126
naming, default. *See* Naming
object, 64, 112
object listing, 68, 76
PACKING.LST, 18
source listing, 67, 68, 73
source/object listing, 67, 68, 78
temporary
SETUP, 20
space required, 481
TMP, 35, 37
Fixups, 141
FL command
canceling, 54
exit codes, 48
file processing, 53
format, 52
/NOD, used with, 56
/NOI, used with, 54
$NOLIST, used with, 69
options, 58
using, 52
Version 4.0, new to, 318
FL exit codes, 48, 339
FL option character
dash (-), 58
forward slash (/), 58
/Fl option (FL), 67
FL options
/4I2 and /4I4, 90, 237
/4Nb, 86
FL options *(continued)*
/4Y6 and /4N6, 91
/4Yb, 44, 86
/4Yb and 4Nb, 217, 237
/4Yd, 44
/4Yd and /4Nd, 87
/4Yf and /4Nf, 44, 91
/4Ys and /4Ns, 91
/4Yt, 272
/4Yt and /4Nt, 91
80186/80188 and 80286 processors, 96
80186/80188 processor, 40
80286 processor, 40
/AH, 59, 225
/AL, 59, 224
/AM, 59, 226
arguments, 58
assembly listing, 67, 247, 262
/c, 43, 63
case, 58
compatibility with Version 3.2, 104
data threshold
default value, setting, 216, 232, 256
/Gt option, 214
moving data items, 217, 223, 228
debug, 44, 86, 217, 237
declare, 44, 87
default integer size, 90, 237
default libraries, 59
differences from LINK options, 123
displaying, 42, 61
external name length, 103
/F, 102, 128, 216, 221
/Fa, 67, 247, 262
/Fc, 67, 262
/Fe, 65
/Fl, 67
floating-point, 195
/Fm, 67
/Fo, 64
FORTRAN 66 programs, 91
/FPa, 59, 120, 195, 199
/FPc
default libraries, overriding, 120
described, 199
example, 43
flexibility, 202
floating-point operations, 59, 195

FL options *(continued)*
/FPc87
8087/80287 coprocessor, 238
default libraries, overriding, 120
described, 198
floating-point operations, 59, 195
/FPi, 59, 195, 199
/FPi87, 59, 195, 198, 238
free-form programs, 44, 91
/Fs, 67
/G0, 96
/G1, 40, 96
/G2, 40, 96
/Ge, 100
/Gr, 104
/Gt
DATA segment, used in, 256
described, 214, 232
FAR attribute, compared with, 228
large and huge memory models, 216
medium memory model, 217
standard memory models, 223
/H, 103
/HELP, 42, 61
/I, 81
include files, searching for, 81
labeling object files, 103
line numbers, 94
line size, 70
/link, 52, 56
memory model
/A options, 59
huge, 225
large, 224
medium, 226
metacommands, used with
$DEBUG, 86
$DECLARE, 88
$DO66, 92
$FREEFORM, 92
$LINESIZE, 71
$NODEBUG, 86
$NODECLARE, 88
$NOFREEFORM, 92
$NOTRUNCATE, 92
$NOTSTRICT, 92
$PAGESIZE, 71
source-file syntax, 91
$STORAGE, 90

FL options *(continued)*
metacommands, used with *(continued)*
$STRICT, 92
$SUBTITLE, 72
$TITLE, 72
naming
executable files, 65
modules, 233
object files, 64
text segments, 222, 223, 233, 234
/NM, 233
/NT, 222, 223, 233, 234
/O, 97
object listing, 67
/Od, 94, 98
/Op, 99, 265
optimization
consistent floating-point results, 99
default, 97
described, 97
disabling, 94, 98
favoring code size, 97
listed, 97
maximum program speed, 97
removing stack probes, 100
SI and DI registers, 265
order on command line, 58
/Os, 97, 265
/Ot, 97
/Ox, 97
page size, 70
preparing for debugging, 43, 98
/Sl, 70
source files, specifying, 62
source listing, 67
source/object listing, 262
/Sp, 70
/Ss, 44, 72
/St, 44, 72
stack size, setting, 102, 216, 221
strict syntax, 91
subtitle, 44, 72
suppressing
compilation, 43, 63
library selection, 101, 202
syntax errors, identifying, 90
/Tf, 62
title, 44, 72
truncating variable names, 91, 272
/V, 103

FL options *(continued)*
/W0 and /W1, 89
warning level, 89
/X, 81
/Zd, 94, 126
/Zi, 43, 94, 98, 134
/Zl, 101, 202
/Zs, 90
FL.EXE file, 35, 36
FL.HLP file, 61
$FLOATCALLS metacommand, 204
Floating point
exceptions
control word, 360
disabling, 358
error messages, 448
listed, 358
status word, 359
operations, optimizing for consistency in, 99
options
compatibility, 203
default, 198
default libraries, 59, 196
function calls, 198, 200
in-line instructions, 198, 200
maximum efficiency with coprocessor, 198
maximum efficiency without coprocessor, 199
maximum flexibility, 203
maximum precision with coprocessor, 198
selecting, 59, 195
SETUP, 29
precision, Versions 4.0 and 3.3, differences, 311
/Fm option (FL), 67
/Fo option (FL), 64
Format specifiers, 240
FORMAT statement, maximum number per program, 482
Formatted I/O, 240
FORTRAN
calling conventions, 263
exit codes, 341
return-value conventions, 263
FORTRAN, books on, 13
fortran keyword (C), 270, 273
FORTRAN.LIB, 239
Forward slash (/)
FL option character, 58
LINK option character, 121
/FPa option (FL), 59, 195, 199
/FPc option (FL)
described, 199
example, 43
flexibility, 202
floating-point operations, 59, 195
/FPc87 option (FL), 59, 195, 198, 238
/FPi option (FL), 59, 195, 199
/FPi87 option (FL), 59, 195, 198, 238
Frame number, 139
$FREEFORM metacommand, 92, 334
Free-form programs, 321
/Fs option (FL), 67

/G0 option (FL), 96
/G1 option (FL), 40, 96
/G2 option (FL), 40, 96
/Ge option (FL), 100
Global symbols. *See* Public symbols, listing
/Gr option (FL), 104
Groups
DGROUP, 131
linking procedures, used in, 141
/Gt option (FL)
_DATA segment, used in, 256
described, 214, 232
FAR attribute, compared with, 228
large and huge memory models, 216
medium memory model, 217
standard memory models, 223

/H option
EXEMOD, 186
FL, 103
/HE option. *See* LINK options, /HELP
Heap, 255
/HELP option
FL, 42, 61
LINK, 123
/help option. *See* /HELP option
/HI option. *See* LINK options, /HIGH
/HIGH option (LINK), 132, 133

Huge
  addresses
    arrays, 217
    defined, 211, 214
    huge model, 225
  arrays, mixed-language
      programming, 288
  memory model
    See also Memory models
    adjustable-size arrays, 219
    /AH option (FL), 225
    assumed-size arrays, 219
    described, 215
    mixed-language programming, 267
HUGE attribute
  alternative to huge model, 225
  common blocks, 219
  default data segment, 214, 216, 217
  effects, 230
  large model, 225
  library routines, used with, 232
  medium model, arguments in, 220,
      226, 227
  using, 228
HUGE_BSS class, 257
Hyphen (-), FL option character, 58

/I option
  FL, 81
  LINK. *See* LINK options,
      /INFORMATION
  MAKE, 173
IF ERRORLEVEL (DOS command), 48,
    86
Ignoring
  case (LINK), 127
  default libraries (LINK), 121, 127
Include files
  nesting, maximum level of, 482
  search path, 81
  standard places, 36, 81
INCLUDE variable, 36, 81
Inexact exception, 448
Inference rules, 177
Infinities, 194
Infinity arithmetic modes, 361
/INFORMATION option (LINK), 124
In-line instructions, 198, 199
Installing the compiler software, 19
Instruction set
  8086/8088 processor, 96
  80186/80188 processor, 96
  80286 processor, 96
Integers
  default size, setting, 90
  maximum size, 482
  mixed-language programming, 279
INTERFACE statement, 270, 271, 273
Internal arithmetic modes, 361
Intrinsic functions
  LOC, 276, 294
  LOCFAR, 276
  Version 4.0, new, 331
I/O
  buffers, 217, 241
  formatted, 240
  list-directed, 240, 314
  Versions 4.0 and 3.3, differences, 324
Italics, 9

Keywords
  calling conventions, specifying, 268
  FORTRAN, 10
  languages, other, 10
  mixed-language programming, 269

Labeling object files, 103
Large memory model
  adjustable-size, assumed-size arrays,
      218
  /AL option (FL), 224
  described, 215
  mixed-language programming, 267
Large memory model. *See* Memory
    models
$LARGE metacommand, 231
LCWRQQ routine
  declaration, 363
  masking denormals, 363
  user's control word, 363
/LI option. *See* LINK options,
    /LINENUMBERS
LIB
  addition commands, 148
  backup library file, 149
  change methods, 149
  combining libraries, 151, 158, 160

LIB *(continued)*
consistency checking, 150, 161
default responses, 156
deletion commands, 148
error messages, 467
exit codes, 339
extending lines, 155, 156
extraction commands, 148
library index, 148
library modules
adding, 151, 158, 159
deleting, 151, 159
extracting, 152, 160
extracting and deleting, 152, 160
replacing, 152, 159
listing files, 148, 152, 161
LLIBFORx.LIB, changing, 298
MLIBFORx.LIB, changing, 298
operations, order of, 148
options
page size, specifying, 150, 162
/PAGESIZE, 150, 162
running
command line, 149
prompts, 155
response file, 156
specifying
commands, 151
output library, 153
terminating, 157
variable, 36, 119
LIB command symbols
asterisk (*), 152, 157, 160
minus sign (-), 151, 154, 157, 159
minus sign-asterisk (-*), 152, 160
minus sign-plus sign (-+), 152, 153, 159
plus sign (+)
appending object files, 157, 158, 159
combining libraries, 160
specifying library, 154
using, 151
Libraries
8087/80287, 198, 199, 238
alternate math, 199
backup, 149
changing with LIB, 147, 149, 158
combining, 151, 158, 160
controlling use, 200
Libraries *(continued)*
creating, 147, 158
default
directory, 20
FL options, 59
emulator, 43, 199
extensions, 118, 147, 149, 150
FORTRAN.LIB, 239
huge model, 225
Intel, 147, 160
large model, 225
LIB input, 149
LIB output, 153
listing (LIB), 148, 152, 161
LLIBFOR7.LIB, 198, 238
LLIBFORA.LIB, 199
LLIBFORE.LIB, 199
medium model, 227
memory models, 210
MLIBFOR7.LIB, 198, 238
MLIBFORA.LIB, 199
MLIBFORE.LIB, 199
names in object files, 55
object modules
deleting, 151, 159
extracting and deleting, 152, 160
including, 151, 158, 159
replacing, 152, 159
RAM disk, used with, 40
search path, 36, 119
SETUP
C, choosing compatibility with, 32
floating-point options, choosing, 29
linking with, 307
memory models, choosing, 29
naming conventions, 31
Versions 3.2 and 3.3, choosing compatibility with, 33
specifying
LIB command line, 149, 153
LINK command line, 113
LINK prompts, 115
LINK response file, 116
standard memory models, support for, 223
standard places, 36, 119
suppressing selection, 101
Version 4.0, changes for, 320
Versions 4.0 and 3.3, compatibility between, 315

Library manager. *See* LIB
Limits
  compiler, 481
  linker, 485
Line size, source listings, 70
Line-number option
  FL, 94
  LINK, 126
/LINENUMBERS option (LINK), 126
$LINESIZE metacommand, 71
LINK
  See also LINK options
  alignment types, 138
  default
    command line, 111
    responses, 115
  error messages
    identifying, 85
    listed, 454
  exit codes, 339
  file-name conventions, 118
  groups, 141
  limits, 485
  mixed-language programming, 299
  operation, 137
  running
    FL command line, 104, 109
    LINK command line, 111
    prompts, 115
    response file, 116
  temporary output file, 110, 124
  terminating, 137
/link option (FL), 52, 56
LINK options
  abbreviations, 122, 123
  case sensitivity, 123, 127
  /CODEVIEW (/CO), 134
  compatibility, preserving, 133
  /CPARMAXALLOC (/CP), 129
  data loading, 132
  debugging, 134
  displaying with /HELP (/HE), 123
  /DOSSEG (/DO), 131
  /DSALLOCATE (/DS), 132
  executable-file loading, 133
  /EXEPACK (/E), 125
  FL options, differences from, 123
  /HELP (/HE), 123
  /HIGH (/HI), 132, 133
  ignoring default libraries, 121, 127, 299
LINK options *(continued)*
  /INFORMATION (/I), 124
  line numbers, displaying, 126
  /LINENUMBERS (/LI), 126
  LINK prompts, responding to, 122
  map file, 113, 126
  /MAP (/M), 113, 126
  /NODEFAULTLIBRARYSEARCH (/NOD)
    C and FORTRAN, linking, 299
    described, 127
    object files, used with, 56, 121
    standard libraries, 200, 202
  /NOGROUPASSOCIATION (/NOG), 133
  /NOIGNORECASE (/NOI), 127
  numerical arguments, 122
  order on command line, 121, 123
  ordering segments, 131
  overlay interrupt, setting, 131, 135
  /OVERLAYINTERRUPT (/O), 131, 135
  packing executable files, 125
  paragraph space, allocating, 129
  /PAUSE (/P), 123
  pausing, 123
  process information, displaying, 124
  segments, 130
  /SEGMENTS (/SE), 130
  specifying on LINK command line, 114
  stack size, setting, 102, 128, 216, 221
  /STACK (/ST), 102, 128, 186, 216, 221
  Version 4.0, new, 318
Linker utility. *See* LINK
$LIST metacommand, 69
List-directed I/O, 240, 314
Listing
  FL options, 41, 61
  LINK options, 123
Listing files
  assembly, 67
  LIB, 148, 152, 161
  map, 67, 222
  object, 67
  source, 67
LLIBFOR7.LIB, 198, 238
LLIBFORA.LIB, 199
LLIBFORE.LIB, 199

LLIBFORx.LIB compatibility with future versions of C, 298
LOC intrinsic function, 276, 294
LOCFAR intrinsic function, 276
Logical values, 292
Lowercase keywords, notation, 10
Lowercase letter. *See* Case significance

/M option. *See* LINK options, /MAP
Macro definitions, MAKE, 173
MAKE
  described, 165
  description file, 166
  error messages, 472
  example, 170
  exit codes, 86, 337, 340
  inference rules, 177
  infile, 168
  macro
    definitions, 173
    names, special, 177
  messages, 172
  options
    /D, 173
    /I, 173
    /N, 173
    /S, 173
    using, 173
  outfile, 168
  running, 172
Map files
  code size, 222
  creating, 67, 69, 95, 126
  extensions, 68, 113, 118, 126
  /Fm option (FL), 67, 69
  format, 79
  frame numbers, obtaining, 139
  /MAP option (LINK), 113, 126
  naming with LINK, 113
  /Zd option (FL), 95
/MAP option (LINK), 113, 126
Math packages. *See* Floating-point options
/MAX option (EXEMOD), 186
Maximums
  length of a name, 482
  level of nesting statements, 482
  number of simple variables per subprogram, 482
Maximums *(continued)*
  size of a character constant, 322, 482
Medium memory model
  adjustable-size, assumed-size arrays, 218
  /AM option (FL), 226
  argument passing, 220
  common blocks, 220
  described, 215
  mixed-language programming, 267
  NEAR and FAR attributes, 227
Memory models
  See also Attributes, FAR; Attributes, HUGE; Attributes, NEAR
  adjustable-size arrays, 218
  adjusting, 210
  argument passing, 220
  assumed-size arrays, 218
  common blocks, 219
  default, 197, 209, 215, 216
  defined, 211
  FL options, 59
  huge
    adjustable-size arrays, 219
    /AH option (FL), 225
    arrays, 217
    assumed-size arrays, 219
    default data segment, 216
    described, 215
  large
    adjustable-size, assumed-size arrays, 218
    /AL option (FL), 224
    default data segment, 216
    described, 209, 215
  library support, 223
  medium
    adjustable-size, assumed-size arrays, 218
    /AM option (FL), 226
    argument passing, 220
    common blocks, 220
    default data segment, 217
    described, 215
  mixed-language programming, 267
  options, default libraries, 59
  selecting, 223, 224
  SETUP, 29
  specifying, 209
  standard, 209, 215, 223

Memory models *(continued)*
Version 4.0, new, 319
Memory-based disk emulator (RAM disk), 40
See also RAM disk
Metacommands
$DEBUG, 86, 217, 238, 357
$DECLARE, 87, 334
$DO66, 92
$FLOATCALLS, 204
$FREEFORM, 92, 334
$LARGE, 231
$LINESIZE, 71
$LIST, 69
$NODEBUG, 87
$NODECLARE, 88, 334
$NOFLOATCALLS, 204
$NOFREEFORM, 92, 334
$NOLIST, 69
$NOTLARGE, 231
$NOTRUNCATE, 92, 272, 334
$NOTSTRICT, 92
$PAGESIZE, 71
$STORAGE, 90, 238
$STRICT, 92
$SUBTITLE, 72
$TITLE, 72
$TRUNCATE, 92, 334
Microsoft LIB. *See* LIB
Microsoft LINK. *See* LINK
/MIN option (EXEMOD), 186
Minimum allocation value, controlling, 186
Minus sign (−), LIB command symbol, 159
Minus sign-asterisk (−*), LIB command symbol, 152, 160
Minus sign-plus sign (−+), LIB command symbol, 152, 159
Mixed-language programming
advantages, 266
arrays, 285, 288
attributes, 269
Boolean types, 282
calling
C procedures from FORTRAN, 275
conventions, 267
FORTRAN procedures from C, 278
characters, 282
compiler versions required, 266
Mixed-language programming *(continued)*
complex numbers, 291
data, sharing, 294
data types, 278
files, 295
fortran keyword (C), 270, 273, 276
/Gc option (CL), 270, 276
huge arrays, 288
input, 295
integers, 279
keywords, 269
linking, 298
logical values, 292
memory models, 267
output, 295
passing
arguments, 267
strings, C to FORTRAN, 286
strings, FORTRAN to C, 286
pointers (C), 279, 286
procedural arguments, 293
procedure pointers (C), 293
real numbers, 283
return-value conventions, 293
SETUP, 32
stack, use of, 267
strings, 284, 285
structures (C), 290, 292
uses, 266
writing interfaces
C to FORTRAN, 276
FORTRAN to C, 273
writing to the terminal, 295
Mixing modules
Versions 4.0 and 3.2, 317
Versions 4.0 and 3.3, 316
MLIBFOR7.LIB, 198, 238
MLIBFORA.LIB, 199
MLIBFORE.LIB, 199
MLIBFORx.LIB, compatibility with future versions of C, 298
MODE option, Versions 4.0 and 3.3, differences, 310
Modules, naming, 233
module_TEXT segment, 257

/N option (MAKE), 173

Names
  compiler limits, avoiding, 483
  internal
    defined, 265
    maximum length, 482
  length, 271, 321, 482
  maximum number per module, 482
  reserved, 265
  scratch file, Versions 4.0 and 3.3, differences, 310
Naming
  executable files
    default, 112
    FL, 65
    LINK, 112
  map files, 113
  modules, 233
  object files, 64
  segments, 233
Naming conventions
  FORTRAN and assembly language, 265
  mixed-language programming, 271, 272
  object files, 272
NANs, 194
Near addresses
  default data segment, 214, 216
  defined, 211, 212
  medium model, 226
  subprograms, 223
NEAR attribute
  adjustable-size arrays, 219
  alternative to medium model, 227
  assumed-size arrays, 219
  common blocks, 220
  declaring subprograms with, 222
  effects, 229
  huge model, 225
  large and huge models, 216, 220, 226
  library routines, used with, 232
  subprograms, 223
Nesting
  include files, 482
  statements, 482
/NM option (FL), 233
NO87 variable, 204
/NOD option. *See* LINK options, /NODEFAULTLIBRARYSEARCH
$NODEBUG metacommand, 87
$NODECLARE metacommand, 88, 334
/NODEFAULTLIBRARYSEARCH option (LINK)
  C and FORTRAN, linking, 299
  described, 127
  FL command, used with, 56
  object files, used with, 56, 121
  standard libraries, 200, 202
$NOFLOATCALLS metacommand, 204
$NOFREEFORM metacommand, 92, 334
NOG option. *See* LINK options, /NOGROUPASSOCIATION
/NOGROUPASSOCIATION option (LINK), 133
/NOI option. *See* LINK options, /NOIGNORECASE
/NOIGNORECASE option (LINK), 54, 127
$NOLIST metacommand, 69
Notation
  apostrophe, 10
  described, 9
$NOTLARGE metacommand, 231
$NOTRUNCATE metacommand, 92, 272, 334
$NOTSTRICT metacommand, 92
/NT option (FL), 222, 223, 233, 234
NUL, 70, 153
NULL segment, 256, 258, 452
Null-pointer assignment, 452

/O option. *See* LINK options, /OVERLAYINTERRUPT
/O options (FL), 97, 265
Object files
  extensions, 64, 112, 118
  FL command, 53
  labeling, 103
  library, names in, 55
  names in, 272
  naming
    default, 64, 112
    FL, 64
  object modules, difference from, 147
  specifying
    LINK command line, 112
    LINK prompts, 115
    LINK response file, 116

Object files *(continued)*
Versions 4.0 and 3.3, compatibility, 315
Object modules
defined, 147
library
deleting from, 151, 159
extracting and deleting from, 152, 160
including in, 151, 158, 159
listing (LIB), 152, 161
object files, difference from, 147
Object-listing files
creating, 67
extensions, 68
format, 76
/Od option (FL), 94, 98
/Op option (FL), 99
Optimization
code size, favoring, 97
consistent floating-point results, 97, 99
default, 97
disabling, 94, 97
execution time, favoring, 97
FL options, 97
maximum program speed, 97
stack probes, removing, 100
storing frequently used variables during, 262, 265
Optimizing. *See* Optimization
Options, FL. *See* FL options
Options, LINK. *See* LINK options
/Os option (FL), 97
/Ot option (FL), 97
Out/dependent file descriptions, 166
/OVERLAYINTERRUPT option (LINK), 131, 135
Overlays
interrupt number, setting, 131, 135
overlay manager prompts, 136
reducing program size, 239
restrictions, 136
search path, 136
specifying (LINK), 104, 135
Overview, 3
/Ox option (FL), 97

Packing executable files, LINK, 125
PACKING.LST file, 18
Page size
library, 150, 162
source listings, 70
$PAGESIZE metacommand, 71
/PAGESIZE option (LIB), 150, 162
Paragraph space, 129
Parameters. *See* Arguments
Passing arguments
See also Attributes
by reference, 260, 267, 268, 279
by value, 220, 260, 267, 268
medium model, 226
mixed-language programming, 267, 269
varying numbers of, 271
PATH command
AUTOEXEC.BAT file, 38
batch files, 38
environment variable, 17, 38
MAKE, used with, 178
PATH variable, 36, 38
/PAUSE (/P) option (LINK), 123
Placeholders, 9
Plus sign (+)
LIB command symbol
appending object files, 158, 159
combining libraries, 151, 160
Intel, XENIX files, used with, 147
specifying library, 154
using, 151
LINK command symbol, 113, 116
Pointers (C), 269, 279, 283, 286
Practice session, 44
PRN, 70
Procedural arguments, mixed-language programming, 293
Procedure pointers (C), 286, 287, 293
Processors
8086/8088, 96
80186/80188, 40, 96
80286, 40, 96
Program entry point, 118
Program header, inspecting, 186
Program maintainer. *See* MAKE
Public names, 272
See also Public symbols, listing
Public symbols, listing
LIB, 148, 152, 161
LINK, 126

Quotation mark, single ('), 10
Quotation marks (" "), 12

RAM disk
  advantages, 40
  libraries, used for, 40
  temporary files, used for, 37, 40
Real numbers, 283
Record structure
  binary direct files, 352
  binary sequential files, 351
  formatted direct files, 347
  formatted sequential files, 345
  unformatted direct files, 350
  unformatted sequential files, 348
Redirecting error messages, 85, 190
Reference, passing arguments by, 260
References
  long, 142
  near segment relative, 142
  near self relative, 142
  resolving, 127, 141
  short, 142
  unresolved, 141
Registers
  BP, 262, 264
  CS, 212, 213, 262, 266
  DI, 262, 264, 265
  DS
    assembly-language routine, 262, 266
    default data segment, 213, 216
    described, 132
    DGROUP, 258
    near addresses, 213
    near, far, and huge addresses, 212
  ES, 212, 258, 266
  SI, 262, 264, 265
  SS, 212, 258, 262, 266
Relocation information, 138
Response files
  LIB, 156
  LINK, 116
Return codes. *See* Exit codes
Return-value conventions
  FORTRAN, 263
  mixed-language programming, 293
Round control, 361
Run time
  error messages
    described, 434
    floating-point exceptions, 448
    redirecting, 190
    run-time library, 435
  libraries, 147
Running
  LIB
    command line, 149
    prompts, 155
    response file, 156
  LINK
    FL command line, 104, 109
    LINK command line, 111
    prompts, 115
  SETUP, 19

/S option (MAKE), 173
Sample hard-disk setup, 37
SCWRQQ function, 363
Search paths
  environment variables, 37
  include files, 36, 81
  libraries, 36, 119
  overlays, 136
  standard, 34, 35
  temporary files, 37
Segments
  alignment types, 138, 140, 259
  _BSS, 255, 256, 258
  c_common, 256, 258
  class names, 139, 259
  class types, 139
  code, 213
    See also Segments, module_TEXT
  combine classes, 259
  combine types, 140
  combining, 140
  CONST, 256, 258
  _DATA, 256, 257, 258
  data
    default, 213, 232, 234
    naming, 234
  module_TEXT, 257
  naming, 233
  NULL, 256, 258, 452
  number allowed, 130
  order, 131, 139, 254

Segments *(continued)*
STACK, 255, 258
text
default, 234
naming, 234
/SEGMENTS option (LINK), 130
Semicolon (;)
LIB command symbol, 149, 150, 156, 161
LINK command symbol, 111, 115, 116
SET command
AUTOEXEC.BAT file, 38
batch files, 38
environment variables, used with, 17, 37
PATH, compared to, 38
SETENV
error messages, 479
exit codes, 340
utility, 188
SETUP
AUTOEXEC.BAT file, 38
CONFIG.SYS file, 39
disk, 18
floating-point options, choosing, 29
installing on floppy-disk system
3-1/2 inch, 25
5-1/4 inch, 21
installing on hard-disk system, 19
memory model, choosing, 29
mixed-language programming, 32
naming libraries, 31
operations, 18
PACKING.LST file, 18
removing error-message text, 32, 237
rerunning, 29, 33
running, 19
Version 3.3, compatibility, 33
SI register, 262, 264, 265
Single left quotation mark (‘), 10
Single right quotation mark (’), 10
/Sl option (FL), 70
Small capitals, 13
Source compatibility, 309
Source listings, specifying
line size, 70
page size, 70
subtitles, 72
titles, 72
Source-listing files
creating, 67
extensions, 68
format, 73
Source/object-listing files
creating, 67
extensions, 68
format, 78
/Sp option (FL), 70
spawnlp routine (C), 295
Special macro names, MAKE, 177
/Ss option (FL), 44, 72
SS register, 212, 258, 262, 266
SSWRQQ function, 362
/ST option. *See* LINK options, /STACK
/St option (FL), 44, 72
Stack
allocating separately from DS, 220
arguments, order of, 260
changing size, 102, 216, 221
default data segment, used in, 216, 221
large, 220
mixed-language programming, use in, 267
overflow, 452
probes, enabling, 100
size, controlling, 186
STACK class name, 131
/STACK option
(EXEMOD), 186
(LINK)
default data segment, 216, 221
described, 128, 186
/F option, compared to, 102
STACK segment, 255, 258
Stack size
default for C programs, 128
setting, 128
Standard places
include files, 36, 81
libraries, 36, 119
temporary files, 37
Start-up routine, 118, 128, 129
Statements, maximum level of nesting, 482
STATUS option, Versions 4.0 and 3.3, differences, 310
Stopping
compiler (FL), 54

Stopping *(continued)*
library manager (LIB), 150, 157
linker (LINK), 137
$STORAGE metacommand, 90, 238
$STRICT metacommand, 92
Strings
mixed-language programming, 284
passing from C to FORTRAN, 286
passing from FORTRAN to C, 286
storage, 284
substring specifications, 323
Structures (C), 283, 290, 292
Subroutines, maximum number of ENTRY statements, 482
$SUBTITLE metacommand, 72
Subtitles, source listings, 72
Swapping disks
during compiling, 54
during linking, 123
Switches. *See* FL options; LINK options
Symbol table, entries, 483, 484
Syntax
described, 9
errors, 90
System requirements, 4
system routine (C), 295

Tc edit descriptor, 240
Temporary files
default directory, 20
RAM disk, used for, 40
space required, 481
standard places, 37
_TEXT, 234
Text segment
default name, 234
naming, 233
/Tf option (FL), 62
$TITLE metacommand, 72
Titles, source listings, 72
TLc edit descriptor, 240
TMP variable, 37
TOOLS.INI file, 178
$TRUNCATE metacommand, 92, 334
Types
arrays, 285, 290
Boolean, 282
character, 282
Types *(continued)*
complex, 291
double precision, 283, 284
integer, 279
logical, 292
real, 283, 284
strings, 285

Underflow exception, 448
Underscore (_)
C names, used in, 272
FORTRAN 4.0 names, used in, 321
Underscores (__), 265
User's Guide, organization, 6
Utilities
default directory, 20
ERROUT. *See* ERROUT
EXEMOD. *See* EXEMOD
EXEPACK. *See* EXEPACK
library manager. *See* LIB
linker. *See* LINK
SETENV. *See* SETENV

/V option (FL), 103
VALUE attribute, 220
Value, passing arguments by, 260
Variables
bitwise manipulation, 331
environment, 34, 35
See also Environment variables
PATH, 38
simple, maximum number per subprogram, 482
VARYING attribute, 271
Vertical bar (|), 11
VM.TMP file, 110, 124

/W0 and /W1 options (FL), 89
Warning error messages
compiler, 426
described, 373
setting level of, 89
Wild-card characters, DOS, 42, 53

/X option (FL), 81

Z edit descriptor, 333
/Zd option (FL), 94, 126
/Zi option (FL), 43, 94, 98, 134
/Zl option (FL), 101, 202
/Zs option (FL), 90

MICROSOFT®

16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem (Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

Documentation Feedback

Please help us make our documentation better by completing and returning this card. Your feedback will help us improve future products.

Name _____

Address _____

City _____ State _____ Zip _____

Product _____ Version no. _____

Operating system _____

Computer _____ Memory (K) _____

RAM disk ☐ yes ☐ no Hard disk ☐ yes ☐ no

Programming experience: _____ years; in this language: _____years

Please rate the documentation on a scale of 1 to 5.

	5	4	3	2	1	
Complete	☐	☐	☐	☐	☐	Incomplete
Accurate	☐	☐	☐	☐	☐	Inaccurate
Organized	☐	☐	☐	☐	☐	Disorganized
Usable	☐	☐	☐	☐	☐	Unusable
Readable	☐	☐	☐	☐	☐	Unreadable
Understandable	☐	☐	☐	☐	☐	Unintelligible
Attractive	☐	☐	☐	☐	☐	Unattractive
Excellent	☐	☐	☐	☐	☐	Poor

Was installation easy to understand? ☐ yes ☐ no

Did you work through the demonstration session? ☐ yes ☐ no

If yes, was it useful? ☐ yes ☐ no

Is information easy to find? ☐ yes ☐ no

Do you use your documentation mainly for:
☐ reference? ☐ learning to program?
☐ learning to operate the software?

Did you find factual errors in the documentation? ☐ yes ☐ no
If yes, please give manual name, page number, and description of the error.

Is more information needed on how to program? ☐ yes ☐ no
If yes, on what topics?

Is the documentation intimidating? ☐ yes ☐ no

Are the manuals:
☐ too long? ☐ too short? ☐ about the right length?

Other suggestions or comments? (Use back of card if necessary.)

Comments and suggestions become the property of Microsoft Corporation.

detach here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 108 BELLEVUE, WA U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT®
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Attn: Manager, Languages User Education
Bldg. 2

04/86

tape here

MICROSOFT®